The PDES-MAS Routing Simulation

Roland Ewald

August 26, 2005

This document is intended as a short documentation [1] of the PDES-MAS routing simulation and visualization tool.

1 Simulation Tool

1.1 Installation

The simulation kernel requires a correctly installed Java Runtime Environment 1.4.2 or better. After downloading the file, no other installation steps are required.

The simulation is deployed as a Java Archive file, which is basically a ZIP file with some additional Java-specific information. By renaming it to a ZIP file you are able to unzip the file with any common packing tool, so that you can have a look at the source code.

1.2 Start

To start the batch mode execution, change to the directory in which the simulation tool JAR file is situated, and enter

```
java -jar pdesmas_sim.jar
```

1.3 Usage

The use of the simulation tool is very easy, just load a parameter file and start the simulations (or a single simulation, that depends on the parameters given in the parameter file). The upper progressbar depicts the progress of the simulation run that is being calculated in the moment, while the one beneath shows the progress of the whole experiment.

After the simulation is finished, the results are collected and written in result files. The files will be stored in a new subdirectory, named after the experiment name (which can be defined in the parameter file).

NOTE 1: If you load a parameter file and an error occurs, it is useful to have a look at the Java console, so that can be seen which parts of the parameters have been rejected (because they are not set in a meaningful way - e.g., a

[1] If any questions occur (which is very likely, since this document does not cover all details), feel free to contact me. My email address is roland.ewald@uni-rostock.de .

negative number of time steps).

NOTE 2: For each simulation run, the simulation counts the number of actions which had to be dropped. An action has to be dropped if the agent wants to access a certain SSV of a spatial type by ID, but there is none in its range (i. e., range reads will never be dropped). Therefore, a high amount of dropped actions just means that some agents have been forced to be less active than defined (by their number of generated events per time step - see section 1.4). This may hamper the simulation results, depending on the objective of the experiment, but it only affects the number of generated ID read or write queries.

NOTE 3: To facilitate the usage of this tool, experiments can be executed automatically one after another. To do so, you just have to open multiple parameter files in the 'Load parameter file...' dialogue.

1.4 Parameters

In this section, the special focus will be on the parameters of the underlying model, their possible values and their relations among each other. A complete reference of all parameters that are available is given in table 1 - 5.

1.4.1 Parameters of the application - model

Number of ALPs, Duration of the Simulation The number of agents (i. e., ALPs) is arbitrary. An ALP can be added to any CLP that is a leaf by setting its parameter `clpNum` to the corresponding CLP - number. Furthermore, the duration of the simulation can be set by changing the `timeStep` parameter. For each time step, an agent can perform certain actions. The amount of actions an agent can perform per time step depends on `eventsPerStepMean` and `eventsPerStepDev` (see table 3). After all agents performed their actions for a time step, the load balancing algorithm is activated and may move shared state variables.

SSV Types The number of SSV types is not restricted. Each type can have a certain value range, defined by `lowerBorder` and `upperBorder`. Of course, $lowerBorder < upperBorder$ has to hold true.

Furthermore, the model distinguishes between so-called *spatial* and *non-spatial* SSV types. A *spatial* SSV type is a type, that the agents within the model will access depending on their location in the modelled space. For example, SSV types which correspond to location attributes of objects, like `Tile::PosX`, are usually spatial types.

Each SSV type which does not have this property is a *non-spatial* SSV type. Those will be accessed by the agents without taking into account their actual location (see section 1.4.1 for details). Besides that, each spatial SSV type has to be attached to (until now) exactly one dimension of the environment, which is represented by the `dimension` parameter. The `valueDistribution` of an SSV type defines the distribution of the SSV values in the beginning of the simulation.

To adjust the values of the initial SSVs, each SSV type has a `valueDistribution` - Parameter. This probability distribution will be used to create the values of

the initial SSVs randomly.

ALP Parameters Usually, situated agents in a MAS have also the ability to move within their environment, so that their access patterns (i. e., the properties of the requests they generate) will change over time. Furthermore, agents do not move randomly in most cases, but with deliberation. Thus, this aspect had to be taken into consideration.

Each modelled agent has to choose a destination at a certain distance. Afterwards, it walks stepwise the shortest way towards its destination, altering each coordinate of its position by a dimension-specific value, `stepSize`, that was set when the target was chosen. When an agent reaches its destination, it picks a new target randomly. The position of the new target will be set at a certain distance, the `targetDistance`, which is the number of steps the agent will need to reach it. During this process, a new `stepSize` is generated randomly for each dimension. However, some of the `stepSize` values may have to be adjusted, in order to realise a linear movement of the agent. Before each step, an agent can generate a number of requests. A request can be a range query or a query for reading or writing a single SSV. To generate range queries, each agent has a certain `range` for each dimension. A generated range query will query an interval of the size `range`, which surrounds the position of the agent in the corresponding dimension.

Number and Properties of Initial SSVs The number of SSVs that already exist in the CLP tree can be defined by `numOfInitialSSVs`. `SSVTypeInitP` is a vector $SSVTypeInitP = (p_1 \dots p_n)^T$ which stores the probability, that a SSV created during initialization belongs to the SSV type i, at the i-th position. Hence, $\sum_{i=1}^{n} p_i = 1$ has to hold true.

1.4.2 Parameters of the simulation - model

Size of the CLP tree The size of the CLP tree can be set implicitly by its number of levels, since the CLP tree is a (complete) binary tree. The parameter `numOfTreeLevels` can be used to do so, the overall number of CLPs in each tree is $2^{numOfTreeLevels} - 1$, and $2^{numOfTreeLevels-1}$ of those CLPs are leaves of the tree and should be used as server CLPs for ALPs.

The CLPs are numbered depending on their position in the tree, level by level and from left to right. This means, the root CLP has the number 0, its left child number 1, its right child number 2, and so on.

Root CLP Imbalance In order to observe different distributions of SSV quantities per CLP at the initial state, the parameter `rootImbalance` can be used. Any value in $[0, 1]$ can be assigned to it. $RootImbalance = 0$ means that the number of SSVs per CLP is (as much as possible) the same for all CLPs in the tree, while $RootImbalance = 1$ implies that all SSVs are hosted by the root CLP. In other words, $rootImbalance$ defines the percentage of all SSVs which will be assigned *additionally* to the root CLP.

Thus, the average number of SSVs per CLP, $|SSV_{CLP}|$, can be calculated by ($C = (c_0, \dots, c_n)$ is the set of all CLPs, the aforementioned CLP numbering is used, so c_0 is the root CLP):

$$|SSV_{c_i, i \neq 0}| = numOfInitialSSVs \cdot \frac{1 - rootImbalance}{|C|}$$

Analogue, the number of SSVs in the root CLP can be calculated by:

$$|SSV_{c_0}| = numOfInitialSSVs \cdot (1 - (|C| - 1) \cdot \frac{1 - rootImbalance}{|C|})$$

Value Fluctuation per Type and CLP Furthermore, there is a possibility to define an initial (CLP-)*fluctuation* for each type. This is a factor in $[0, 1]$, that constraints the value range of SSVs on one CLP.

It defines the maximum difference between the values of two SSVs of this type and on this host as a ratio of the value range of that type. This means, each CLP c_i has to fulfill this unequation for each type t:

$$\max(SSV_{c_i,t}^{value}) - \min(SSV_{c_i,t}^{value}) \leq fluctuation \cdot (t_{upperBorder} - t_{lowerBorder})$$

1.4.3 Parameters of the simulation

There are only very few parameters to adjust the simulation process, the most important being the `name` of an experiment. The name determines the names of the result files and the name of the local directory they are stored in. Moreover, it is possible to select only certain results from the routing algorithms (`saveResults`) and the load balancing algorithm (`saveLBResults`).

For each experiment, two parameters can be chosen which will be varied. Each parameter will be varied between a `lowerBorder` and an `upperBorder`. The number of variations is determined by the `resolution`. Each combination of the varied parameters will be used to parameterise a single simulation run. Thus, if the first parameter has a resolution of 10, and the second one has a resolution of 40, $10 \cdot 40 = 400$ simulation runs will be executed. Therefore, the result matrices will have 10 rows and 40 columns (see section 1.7).

NOTE: Not only single values can be varied, but also arrays with up to two dimensions. This makes it possible to vary the behaviour of agents, which is described by matrices, consistently (i. e., the sum of all access probabilities will be 1 for each variation between $(1, 0, 1, 0, 0)$ *and* $(0, 1, 0, 1, 0)$*).*

Besides those general parameters, there are two flags which control the creation of additional log files if (and *only* if) the experiment consists of one simulation run (i. e., the resolutions of the varied parameters are both set to 1): `writePEL` causes the generation of a PEL file (which can be visualized with the visualization tool described in section 2), whereas `writeAgentLog` results in the creation of a matrix containing movement information about the agents. In this matrix, each row contains the position of all agents for a certain time step (first row = first time step). Thus, the matrix has columns for each agent's location values. For example, in a two dimensional environment, the first and second column would contain the positions X and Y of agent 1, etc. .

1.5 Parameter Summary

This section summarises all parameters in table 1 - 5.

Name	Description
`timeSteps`	Duration of the simulation
`numOfInitialSSVs`	Number of shared state variables in the initial state
`SSVTypeInitP`	Probabilities, that a variable created for the initial state has a certain SSV type. (*NOTE:the length of the array has to correspond with the number of SSV types*)
`numOfEnvironmentDims`	Number of dimensions of the environment the agents are situated in
`ssvTypes`	Array of all existing SSV types. See table 2.
`alpParameters`	Array of all existing ALPs. See table 3.

Table 1: Parameters for the application model

`spatial`	If true, this is a spatial SSV type
`dimension`	Environment dimension to which this SSV type is attached (implementation for multiple dimensions etc. is not ready yet). Will only be used for spatial SSV types.
`lowerBorder`	Minimum value that SSVs of this type can have
`uppperBorder`	Maximum value that SSVs of this type can have
`fluctuation`	Defines the maximum value fluctuation of SSVs of this type per CLP (for initialization).
`valueDistribution`	Characterises the general value distribution for SSVs of this type. Possible Distributions are. See javadoc documentation for details on implemented distributions and their properties.

Table 2: Parameters for a type of shared state variables (`ssvType`)

Note 1: Serialized Java Beans support referencing. This means, the parameters for the first ALP can be used as a 'default parameters' by setting the parameters of the other ALPs to references. However, this is not possible for primitives (i. e., `int`, `double`, etc.), since they are not regarded as objects and can only be saved by value. The default parameter files are generated in that way. Thus, to change the `accessBehaviour` of all ALPs, only the properties of the first ALP have to be saved. However, if editing the `clpNum` or the `eventsPerStepMean` properties, all ALP parameters have to be changed.

Note 2: When speaking of mean and deviation parameters, a normal distribution is used to generate random numbers which are based on these settings.

Name	Description
accessP	an array that stores the probability of being accessed by this agent for each SSV type. The sum of the probabilities in this array should be 1.
accessBehaviour	Defines the kind of queries the agent generates. This is a two-dimensional array: Each row stores the probabilities of query types for a certain SSV type. The columns store the probability of a range-read, an id-read, a write, an add or a delete query (in this order). The sum of all probabilities of one row has to be 1.
clpNum	The number of the server CLP for this ALP.
eventsPerStepMean	Mean number of events this agent will generate per time step.
eventsPerStepDev	Deviation of number of events this agent will generate per time step.
rangeMean	Mean range of this agent per SSV type.
rangeDev	Deviation of this agent's range per SSV type.
stepSizeMean	Mean step size per environment dimension.
stepSizeDev	Deviation of step size per environment dimension.
targetDistanceMean	Mean target distance (in number of steps needed to reach it).
targetDistanceDev	Deviation of target distance.
writeValueOffsetDistributions	Distribution of offsets for updated values per type.

Table 3: Parameters for the agent model

1.6 Specification of Parameters

To specify the parameters for an experiment, an XML-file has to be created. This will be converted to a Java-Object that parameterises the whole system. To distinguish those files from other XML files, they should be saved with the file ending 'pxml'.

The creation of a correct file from scratch is very difficult, due to the large number of parameters and the relatively complex structure of a serialized Java Bean file (see [2] for details). However, `default_params.pxml` is a default parameter file which can be used to create similar experiments by just editing the values of the parameters that need to be changed. In this file, all parameters are commented, so this should be self-explanatory.

Besides using the commented default parameter file, the simulation tool creates a default parameter file automatically when it is started. If one adjusts the

Name	Description
`numOfTreeLevels`	Number of levels in the CLP tree
`rootImbalanceBonus`	Imbalance bonus for the root name in the initial state (1: all SSVs are located in the root CLP, 0: SSV are distributed uniformly over all CLPs)
`loadBalancing`	Defines an algorithm class used for Load Balancing.
`routingAlgorithms`	Array of routing algorithms that have to be evaluated.

Table 4: Parameters for the simulation model

code of the function `getDefaultParameters()` in `pdesmas_sim.execution.SimulationExecution` (or just the default parameters it uses), it is very easy to create new experimental setups semi-automatically.

Below, an sample snippet of a parameter file is given. The explanations for each variable can be found in table 1 - 5.

Listing 1: Sample snippet of a PXML file

```
<void property="randSeed">
 <long>0</long>
</void>

<void property="numOfInitialSSVs">
 <int>12400</int>
</void>

<void property="numOfTreeLevels">
 <int>5</int>
</void>
```

1.7 Format of Results

Each result matrix is stored in a single tab-separated file (for easy import into Matlab, Excel, etc.). The first varied parameter was changed row by row, the second one column by column. To obtain the result for the upper left edge of the matrix, both parameters had been set to their `lowerBorder` values. Consequently, the bottom right edge contains the result of the simulation run for which both parameters were set to their `upperBorder` values.

2 CLP Log Visualizer

2.1 Installation

There is no further installation needed. See section 1.1 for further details.

Name	Description
`name`	Name of the experiment, determines names of result directory and files.
`writePEL`	If true, an action log will be attached to the log file (for visualization purposes). Will be turned off automatically if more than one run is executed (to increase simulation speed).
`writeAgentLog`	If true, a matrix describing the movement of the agents is written to `[name]_agent_positions.tab` (in the results directory).
`saveResults`	List with all measurements to save as results. For each (routing) algorithm, the data for each result type will be stored to a single file, which is added to the result directory. If an algorithm does not provide a requested result type, the corresponding file will be omitted.
`randSeed`	Initialization value for random number generator (0 means, the current UNIX time will be used)
`dim1FieldName`	Specifies the name of the first parameter to be varied. If the field belongs to a sub-object (like ssvType, alpParameters or routingAlgorithms), use '.' to divide them, e. g. `sssvType.fluctuation` .
`dim1Resolution`	Resolution for first parameter
`dim1LowerBorder`	Start value for first parameter
`dim1UpperBorder`	End value for first parameter
`dim2FieldName`	see `dim1FieldName`
`dim2Resolution`	see `dim1Resolution`
`dim2LowerBorder`	Start value for first parameter
`dim2UpperBorder`	End value for first parameter

Table 5: Parameters for the simulation execution

2.2 Start

To start the CLP log visualization, change to the directory in which the simulation file is situated, and enter:

```
java -cp pdesmas_sim.jar clptreeviz.MainWindow
```

This should start the user interface for the logfile visualization. This tool can visualize PDES-MAS-simulation execution log files (*.PEL). PEL files can be generated by the PDES-MAS routing algorithm simulation.

2.3 Usage of the animation canvas

Because the Batik [1] Toolkit was used to generate the animations, there are no additional functions to change the view of the animation panel. Instead, the Batik commands should be used, as described in table 6.

Keys	Function
SHIFT + Left Mouse Button	Pan
SHIFT + Right Mouse Button	Zoom
CTRL + Right Mouse Button	Rotate

Table 6: Batik commands

2.4 Applet view

The visualization tool can also be used as an applet. The class that wraps the whole user interface is called `clptreeviz.CLPTreeVizApplet`. The parameter `pelFileURL` provides the Applet with a URL to a PEL file. The PEL file will be loaded and displayed.

Below, an example of how to include the applet into an HTML file is given.

Listing 2: Sample usage of the visualization applet in an HTML file

```
<html>
<head>
    <title>PDES-MAS Routing Simulation Visualization</title>
</head>

<body>

<applet
code="clptreeviz.CLPTreeVizApplet"
archive="file:///E:/java_workspace/clptreeviz/pdesmas_sim.jar"
width="800"
height="600">

<param
name="pelFileURL"
value="file:///E:/java_workspace/clptreeviz/example.pel"/>

</applet>

</body>
</html>
```

2.5 PEL file format

2.5.1 General structure

A PEL file consists of four major parts. These parts are divided by lines that start with a '#'. All values in a line are separated by ';'. All integer numbers used to identify CLPs, ALPs etc. are starting from 0, i. e. a C-style enumeration was chosen.

The sequence and content of the parts is as follows:

1. The first part has no special structure. It can be used to store general information concerning the circumstances under which this execution log

was generated, such as date and name of the PXML file, or certain parameter settings.

2. Secondly, the most important parameters (e. g., number of ALPs, structure of the CLP tree) are defined.
3. The third part describes the initial state of the CLP tree as a list of shared stated variables.
4. The last (and usually the by far largest) part is formed by the sequence of *actions* (e. g., a range query or a SSV movement), that occurred during the simulation run.

2.5.2 Definition of general parameters

The definition of general parameters is very short, they are all written in one line. The order of the parameters is as follows:

1. Number of levels in the CLP tree
2. Number of SSV types
3. For each server CLP, the number of ALPs that are connected to it

2.5.3 Description of the initial state

The description of the initial state is given by the definition of SSVs, including their position, their value and their type. Each line stores the definition of one SSV and has the structure:

1. (Unique) ID of the shared state variable.
2. SSV type of the variable, given by an integer.
3. Value of the variable, a real number.
4. Position of the hosting CLP. Numbers are assigned to CLPs top-down and from left to right. So, the root CLP has the number 0, his left child the number 1, his right child the number 2, etc. .

2.5.4 The action log

Since there are many different types of actions, these will be described in detail. Generally, each action is stored in a single line. The first character of the line determines the type of the action. Until now, there are three different actions, to visualize *Load Balancing*, *Queries* and *Routing*. The next paragraphs specify the structure of each action definition.

Load Balancing Actions

1. 'L', to mark this action as a load balancing action
2. ID of the SSV to be load balanced
3. Old SSV position (a CLP - Number)
4. New SSV position (a CLP - Number)

Queries All queries are marked as such by a 'Q' as a first character of the line. Moreover, each query needs a unique query ID and the number of the ALP which generated the query:

1. 'Q'
2. (Unique) Query ID
3. Number of the ALP that generated this query

The rest of the line differs according to the query type, which is set by the following character.

Range Read Queries

1. 'R'
2. Number of SSV type
3. Lower border
4. Upper border

ID Read Queries

1. 'I'
2. ID of the SSV to be read

Write Queries

1. 'W'
2. ID of the SSV to be updated
3. New value

Add Queries

1. 'A'
2. ID of the SSV to be added
3. SSV type
4. Value
5. Position

Delete Queries

1. 'D'
2. ID of the SSV to be deleted

Routing Costs Each simulated algorithm can save its routing cost for each query. The routing performance of a certain algorithm is stored in one line per query:

1. 'R' marks these lines as routing 'actions'
2. ID of the query associated with this routing cost calculation
3. Acronym of the used routing algorithm (until now, 'ABR'(address-based routing), 'RBR'(range-based routing), 'RBI'(range-based routing with range-based ID queries) and 'OPT'(optimal routing) are defined)
4. Number of messages
5. Number of hops
6. A list of edge numbers to show the parts of the CLP tree where communication occurred. Instead of semicolons, these numbers are separated by space characters. The edges are numbered top-down and from left to right, like the CLPs. Thus, the number of the edge can be calculated by the number of the adjacent child CLP - 1. The number of occurrences of an edge number is determined by the amount of messages which are passed through this connection. Thus, an edge number that occurred twice means that two messages have passed this connection in order to resolve the query. The order of the edge numbers is irrelevant.

2.5.5 Example

This example shows parts of an existing PEL file. The **bold-faced** lines are good examples for the described structures.

Listing 3: Sample PEL File

```
Parameter loaded on: Fri Aug 19 13:50:47 BST 2005
Number of swapped variable pairs: 0
Number of swapped variable pairs: 0
Number of swapped variable pairs: 0
Number of swapped variable pairs: 0
Experiment started: Fri Aug 19 13:50:47 BST 2005
Results are stored in: test_lb\test_lb_results.log
#### Start of PEL
4;4;1;1;1;1;1;1;1;1;
#### Initial state of CLP tree
38;2;91.88069704258278;0
57;2;58.18400725005658;0
94;0;59.96331469623942;0
23;2;95.37487834316258;0
62;3;96.28339536924848;0

[...]

#### Action log
```

Q;1464;2;R;2;40.30040495612424;45.30040495612424
R;1464;ABR;6;6;0 3 8 0 3 8
R;1464;RBR;28;12;8 3 0 1 4 10 11 5 12 13 2 6 7 9 8 3 0 *[...]*
R;1464;RBI;28;12;8 3 0 1 4 10 11 5 12 13 2 6 7 9 8 3 0 *[...]*
Q;1465;3;W;31;21.081201648054716
R;1465;ABR;5;4;9 3 0 0 1
R;1465;RBR;3;3;9 3 0

[...]

L;30;0;2
L;31;0;1
L;1;0;2
L;3;0;2

[...]

3 Technical Details

Both tools use the same basic set of classes to represent actions, shared state variables or the CLP tree. In `pdesmas_sim.jar`, there are several folders that store the Batik classes needed by the visualization tool - only the folders (i. e., packages) `clptreeviz` and `pdesmas_sim` contain the simulation tools.

The main part of the software is organized within the `pdesmas_sim` package:

- `pdesmas_sim.algorithms`: A package for all algorithms, so far only routing algorithms (subpackage `routing`) and load balancing algorithms (subpackage `loadbalancing`) are implemented.
- `pdesmas_sim.execution`: The class `SimulationExecution` executes the simulation runs and calls all algorithms. It also loads the (serialized) classes `Parameters` and `ALPParameters` (stores all parameters for a single ALP). The interface `SimulationObserver` can be used to build other user interfaces for the simulation tool, e. g. an interactive simulation.
- `pdesmas_sim.model`: This is the core package for both tools, containing all action classes (subpackage `actions`), some classes to generate certain probability distributions (subpackage `distributions`) and all other classes that model entities of the simulation. `SSVDB` is the data structure used to store the actual state of the CLP tree (i. e., all shared state variables, their values and their positions).
- `pdesmas_sim.ui`: Package for the user interface of the simulation tool.
- `pdesmas_sim.utils`: Generic utilities.

The structure of the `clptreeviz` package is very simple: the subpackage `icons` stores the icons to be used in the toolbar of the visualization tool, and the subpackage `utils` contains some rendering classes for lists and tables.

The most important classes are situated within in the `clptreeviz` package itself:

- `MainWindow` Main class of the visualization tool, contains the whole user interface. Also creates the basic Batik document, i. e. the SVG elements for CLPs, ALPs etc. .
- `Converter` Converts a PEL file into an `SSVDB` object representing the initial state and a list of all actions (which will be animated).
- `AnimationThread` Animates the CLP tree.

More details can be found in the javadoc documentation.

References

[1] http://xml.apache.org/batik/

[2] http://jdj.sys-con.com/read/37550.htm