RDMSim Exemplar: User Guide

January 2021

1 Introduction

RDMSim exemplar represents a simulating environment for a Remote Data Mirroring (RDM) network [1, 2]. It has been designed to help researchers working in the area of self-adaptive systems (SASs) to validate their approaches. The simulator has been designed keeping in view the operational model of the RDM network presented in [2, 3].

In this document, we provide the details of using RDMSim exemplar to perform experiments with self-adaptive decision-making techniques. The document is organized as follows: The second section contains the details about the RDMSim package and its download details. In third section, we provide the details about the architecture of the RDMSim exemplar. In fourth section, we discuss the details of creating custom adaptation logic for RDMSim network with the help of a simple adaptation example.

2 RDMSim Package

The RDMSim package is available in the form of a zip file containing the source files of the simulator. It can be downloaded from the following git hub repository:

`https://gitlab.com/humasamin/rdmsimexemplar`

The RDMSim package contains two sub directories as follows:

1. Source

The Source directory contains the source code of the RDMSim simulator. The source code can be executed with the help of Eclipse software[1]. The Source directory contains the following projects:

a) RDMNetwork

RDMNetwork project represents the simulator software for the RDM network. It helps in execution of experiments by running simulations for the RDM network.

b) TestRDM

TestRDM project contains a custom adaptation example that uses RDMSim exemplar to perform adaptation experiments.

The Source directory also contains a directory *config_log_files*. The *config_log_files* directory contains the configuration and log files for the RDMSim exemplar. The configuration file can be used for the configuration parameter settings of the simulator during the execution of the experiments. The details of the configuration parameters are provided in 3. The *log* file is used to store the results log of the experiments.

[1] www.eclipse.org

2. Jar Files

The *Jar Files* directory contains ***RDMSim.jar*** a Java Archive File for the RDMSim. It helps in usage of the RDMSim exemplar as a library by other java programs. It also contains the *json-simple.jar* file that is used by the simulator to deal with the configuration files for the RDMSim Exemplar.

3 RDMSim Architecture

The RDMSim exemplar has been developed to facilitate the implementation of a two layered architecture for a self-adaptive RDM network as shown in Fig 1. The layers are described as follows:

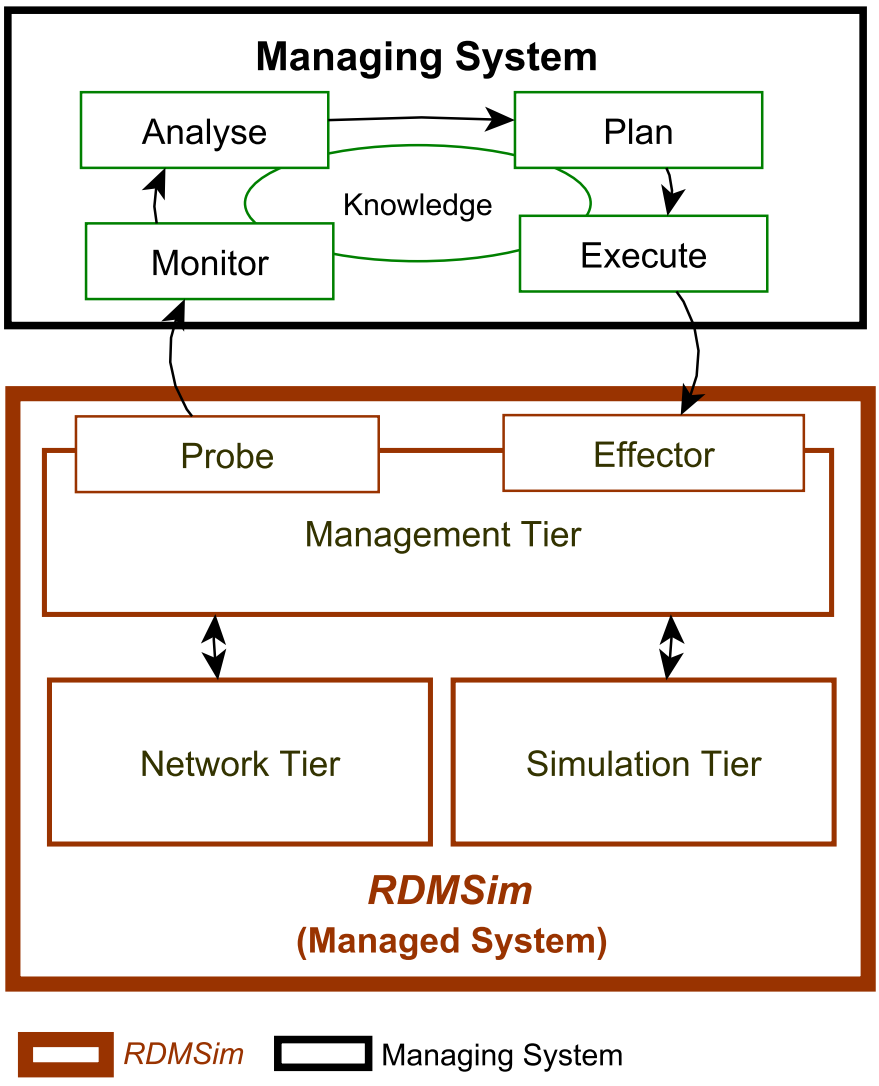

Figure 1: RDMSim Architecture

Table 1: Probe Functions

Function	Description
Topology getCurrentTopology()	returns the current topology for the network.
int getBandwidthConsumption()	returns the bandwidthconsumption of the network.
int getActiveLinks()	returns the number of active links.
int getTimeToWrite()	returns the time to write data for the network.
Monitorables getMonitorables()	returns the values for all the monitorable metrics.

Table 2: Effector Functions

Function	Description
void setNetworkTopology(int timestep,Topology selectedtopology)	to set the network topology at a particular timestep.
void setActiveLinks(int active_links)	to set the number of active links for the network.
void setTimeToWrite(double time_to_write)	to set the time to write data for the network.
void setBandwidthConsumption(double bandwidth_consumption)	to set bandwidth consumption for the network.
void setCurrentTopology(Topology current_topology)	to set topology for the network.

3.1 Managing System

The Managing System layer is responsible for managing the RDM network. The Managing System typically implements the Monitor-Analyse-Plan-Execute over Knowledge base (MAPE-K) feedback loop to perform self-adaptive decisions on the Managed System (RDMSim in our case). The Managing System interacts with the Managed System using probe and effector interfaces. The probes are used to get the monitoring information from the Managed System and the effectors are used to send the adaptation decisions to the Managed System. The RDMSim exemplar provides the interfaces of probes and effectors that can be used by the *external* Managing System to interact with the simulator.

3.2 Managed System

The Managed System is the system being managed by the Managing System. The RDMSim exemplar represents a simulating environment for the RDM network that can be used as a Managed System.

Next, we present the architectural tiers implemented as Java Packages for our RDMSim software. These architectural tiers for RDMSim exemplar are discussed in order of their interaction as shown in Fig 1.

3.3 RDMSim Architectural Tiers

The RDMSim exemplar comprises of the following tiers:

3.3.1 Management Tier

The Management Tier is the tier that provides an implementation of probe and effector interfaces to be used by the external managing system. The Management tier acts as bridge between the Managing System and the Network and Simulation tiers of the RDMSim exemplar to pass on the information related to the monitorable metrics and adaptation decisions.

The functions provided by the probe and effector interfaces can be used to monitor the status of the RDMSim network and change the network topology and various network parameters are described in Table 1 and 2 respectively.

3.3.2 Network Tier

The Network tier provides an implementation of the components of the RDM network. The components implemented by the Network tier are the RDM Network properties comprising of the number of mirrors (servers) and the network links to represent a fully connected network of mirrors. For example, for a

network of 25 mirrors, it will create a network of 300 network links. The Network Tier also provides an implementation of the monitorable metrics and topologies for the network. In the RDMSim exemplar, we provide an implementation of three monitorable metrics as follows:

Total Active Network Links: to measure the reliability for the RDM network.

Total Bandwidth Consumption: to measure the operational cost for the RDM network. It is measured in GigaBytes per second.

Total Time to Write Data to mirrors: to measure the performance of the network in terms of maintaining multiple copies for the network. It is measured in milliseconds.

Considering a synchronous mirroring protocol, the bandwidth consumption and the time to write data is dependent on the number of active links. We compute the Bandwidth Consumption as *Total Bandwidth Consumed=number of active links * bandwidth of one link*[2]. The time to write data is computed as $TotalWritingTime = numberofactivelinks * Timeinmilliseconds$[3].

3.3.3 Simulation Tier

The Simulation tier provides an implementation of the properties of the simulations to be executed by the RDMSim exemplar. The simulation properties includes the number of simulation runs to be executed by the RDMSim network. It also includes an implementation of the uncertainty scenarios representing the different dynamic environmental conditions for the RDM network.

The Simulation tier works at the same level as that of the Network Tier to interact with the Management Tier as shown in Fig 1.

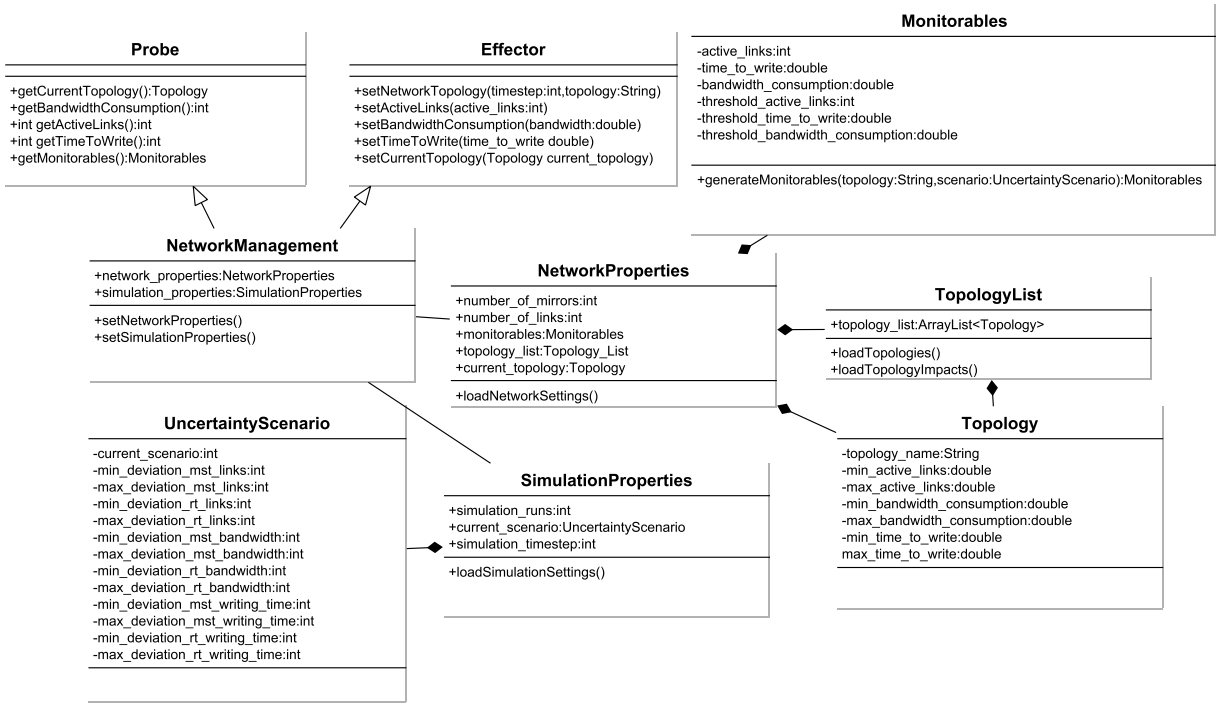

Figure 2: RDMSim Class Diagram

[2]To implement realistic impacts we vary the bandwidth per link between 20 to 30 GBs
[3]To implement realistic impacts, we vary the time between 10 to 20 milliseconds

The class diagram representing the components of the Management, Network and Simulation Tiers is presented in Fig 2. The *NetworkManagement* class along with the *Probe* and *Effector* interfaces provides an implementation of the Management tier. The classes *NetworkProperties*, *Monitorables*, *Topology* and *TopologyList* are part of the Network tier to provide an implementation of the features of the RDM Network. The *SimulationProperties* and *UncertaintyScenario* classes are part of the Simulation tier and are used to implement the functionalities related to the simulations to be executed.

4 Custom Adaptation Example

In order to develop a custom adaptation logic, the RDMSIM simulator provides the interfaces of Probe and Effector. The Probe and Effector functions can be used to implement the MAPE-K feedback loop to support self-adaptation. The Probe helps in monitoring of the data about the number of active links, bandwidth consumption and performance of the network in the form of time to write data. The Effector helps in setting the network topology and tuning of the network settings such as changing number of active links etc.

Next, a step by step example of writing a custom adaptation logic is presented.

4.1 Example

We provide a simple adaptation example that performs adaptations by switching between the topologies of Minimum Spanning Tree (MST) and Redundant Topology (RT) using the effector. The data about the monitorable metrics such as number of active links, bandwidth consumption and time to write is data gathered by using the probing functions.

Step: 1 Create a Java Project

First of all, create a new Java Project in Eclipse IDE using the following steps:

```
Click on File -> New -> Project -> Java Project
```

Name the project as *TestRDM* and Click Finish as shown in Fig 3.

Step: 2 Adding the RDMSim to buildpath of the project

Right click on the "TestRDM" project in the Project Explorer pane – Select *Build Path* – Select *Configure Build Path* as shown in Fig. 4.

A properties dialog box will be displayed as shown in Fig 5.

Go to *Libraries* tab and click on *Add External JARs* and add *RDMSim.jar* to the project from the Jar Files directory of the RDMSim package. Click on *Apply and Close*[4].

Step: 3 Adding Configuration file to Project

Go to the Source folder of the RDMSim package and copy the *config-log-files* folder. Right click on the root of the *TestRDM* project in the Project Explorer pane and paste.

Now, the *RDMSim* exemplar is ready to be used as part of the *TestRDM* project. Next, we describe the step by step usage of the RDMSim exemplar by writing our own adaptation logic.

[4]Please also add the json-simple.jar file as an External Jar

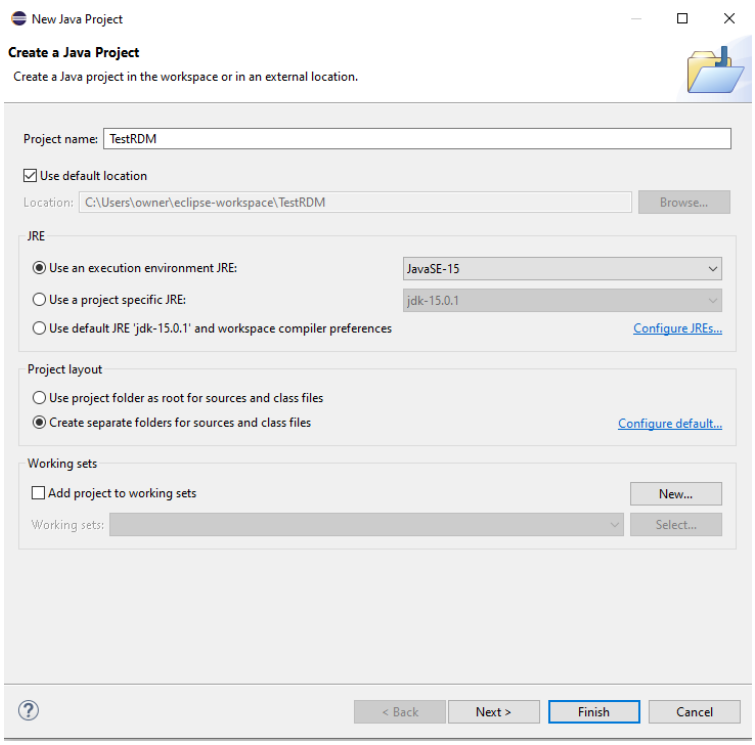

Figure 3: Create New Project

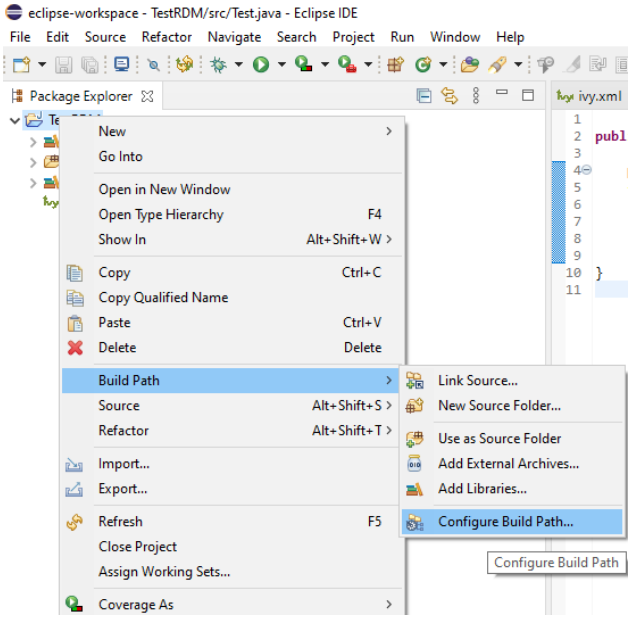

Figure 4: Add RDMSim to the project

Step: 4 Loading Configuration Settings and Instantiation of Probe and Effector

The first step in implementing the custom adaptation logic is to load the configuration settings for the experiment from the *configuration.json* file and instantiation of the Probe and Effector components. The Probe

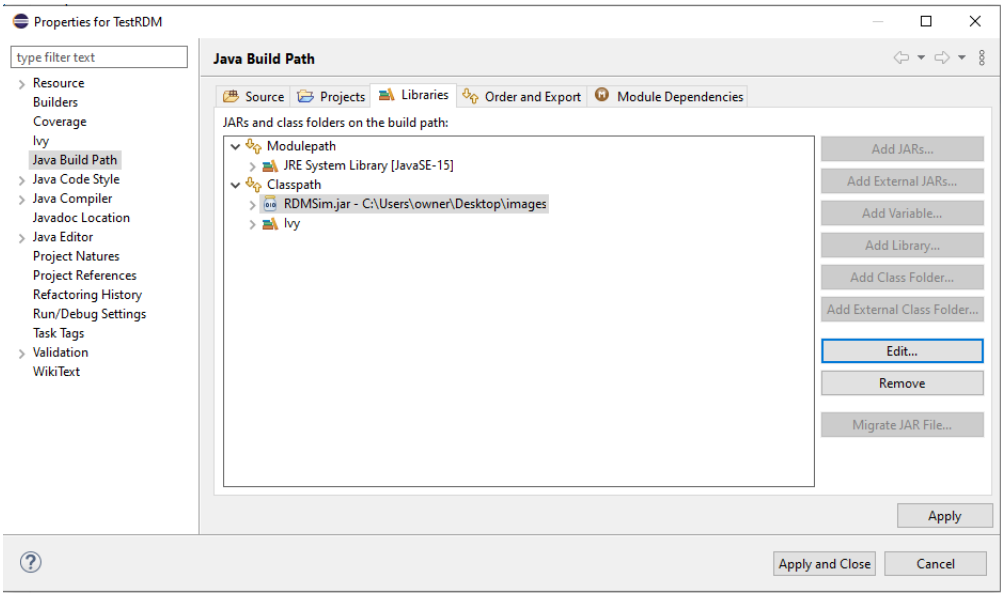

Figure 5: Properties Dialog

and Effector components will enable the communication between our *TestRDM* program and *RDMSim*. This can be done by using the NetworkManagement class in your program as follows:

```
NetworkManagment network_management=new NetworkManagment();
Probe probe=network_management.getProbe();
Effector effector=network_management.getEffector();
```

The NetworkManagement class is responsible for loading the configuration parameters and instantiating the Probe and Effector instances. The configuration settings include the parameters like number of simulation time steps, the number of mirrors for the RDM network, number of active links and uncertainty scenario to be considered for the experiments. The details of the configuration parameters is provided in Table 3.

For this example, we consider an RDM network of 25 mirrors having 300 links in total to create a fully-connected network. We have set the other configuration parameters considering this infrastructure. The values provided in the configuration file can be considered as the default values that are set considering the expert knowledge provided in [3]. These values can be changed according to the experiments' requirements.

Once the Probe and Effector are instantiated, the Probe and Effector functions can be used to monitor the RDMSim network and apply the adaptations using the functions provided in Table 1 and 2 respectively.

Step: 5 Monitoring of the RDMSim network using Probe functions

In order to monitor the RDMSim network, we can use the probe functions provided in Table 1. For example, to get the values of all the monitorable metrics for a particular simulation time step we can use the getMonitorables() function as follows:

```
Monitorables m=probe.getMonitorables();
```

Step: 6 Performing Adaptations on the RDMSim network using Effector functions

In order to perform adaptations on the network, we can use the Effector functions provided in Table 2. For example, to change the network topology at a particular timestep, we can use the setNetworkTopology() function as follows:

Table 3: Configuration Settings

Configuration Parameter	Value Type	**Value Range**	**Description**
time_steps	Integer	0 to Valid Integer Range	It represents the total number of simulation time steps (simulation runs).
mirror_number	Integer	5 to 30000	It represents the number of mirrors in the RDM Network.
link_threshold	Double	0 to 100	Represents the percentage satisfaction threshold for active links.
bandwidth_threshold	Double	0 to 100	Represents the percentage satisfaction threshold for bandwidth consumption.
writing_times_threshold	Double	0 to 100	Represents the percentage satisfaction threshold for time to write data.
topologies	List of String values	[mst,rt]	Represents the list of topology names. We have defined two topologies Minimum Spanning Tree (mst) and Redundant Topology (rt).
topology_active_links	List representing the range: [min, max]	min: 0 to 100 percent of total number of links max: 0 to 100 percent of total number of links	Represents the range (min, max) for the active links for the specific topology. For example: mst_active_links:[min,max]
topology_bandwidth_consumption	List representing the range: [min, max]	min: 0 to 100 percent of total bandwidth max: 0 to 100 percent of total bandwidth	Represents the range (min, max) for the bandwidth consumption for the specific topology.
topology_writing_time	List representing the range: [min, max]	min: 0 to 100 percent of total time to write max: 0 to 100 percent of total time to write	Represents the range (min, max) for the time to write data for the specific topology. For example: mst_writing_time:[min,max]
current_scenario	Integer	0 to 6	Represents the uncertainty scenario to be executed by the RDM Simulator. 0 represents the stable scenario. 1 to 6 represents the detrimental scenarios 1 to 6.
deviation_scenario_scenario_topology_monitorable	List representing the deviation range: [min, max]	0 to 100	Represents the percentage deviation range for the monitorable metric for the current scenario under a specific topology. For example: deviation_scenario_0_mst_links:[min,max]

```
effector.setNetworkTopology(10,"mst");
```

It will set the Minimum Spanning Tree (MST) topology for the network at the simulation timestep 10.

Next, we present a step by step implementation of a simple MAPE-K feeback loop using the code provided in Steps 4 to 6.

MAPE-K loop implementation

Create a Java Class in the TestRDM project and name it MAPE_KLoop using the following steps:

```
Right Click on TestRDM project--> New -->Class
```

The MAPE_KLoop class will be used to implement the phases (Monitor,Analyse,Plan and Execute) of the MAPE-K feedback loop. Copy and Paste the following code in the MAPE_KLoop class.

```
import rdm.management.Probe;
import rdm.management.Effector;
import rdm.management.NetworkManagment;
import rdm.network.Monitorables;
import rdm.network.Topology;

public class MAPE_KLoop {

Probe probe;
Effector effector;

public MAPE_KLoop(Probe probe, Effector effector)
{
this.probe=probe;
this.effector=effector;
}
//Monitor the network using probe functions
public void monitor(int simulation_timestep)
{
Monitorables m=probe.getMonitorables();
analysisAndPlanning(simulation_timestep, m);
}

//Analysis and planning for the adaptation
public void analysisAndPlanning(int simulation_timestep,Monitorables m)
{
String selected_topology;

if (probe.getBandwidthConsumption()>m.getThresholdBandwidthConsumption()
    ||probe.getTimeToWrite()>m.getThresholdTimeToWrite())
{
selected_topology="mst";
execute(simulation_timestep,selected_topology);
}
else if(probe.getActiveLinks()>m.getThresholdActiveLinks())
```

```
{
selected_topology="rt";
execute(simulation_timestep,selected_topology);
}
else
{
selected_topology="rt";
execute(simulation_timestep,selected_topology);
}
}

//Execute the adapatation using functions of the effector
public void execute(int simulation_timestep,String selected_topology)
{
effector.setNetworkTopology(simulation_timestep,selected_topology);

}

public void run(int simulation_timestep)
{
monitor(simulation_timestep);
}
}
```

The phases of MAPE-K loop are implemented as follows:

1. Monitor Phase

We have implemented the monitor() function in the class to implement the monitor phase of MAPE-K loop. The monitor() function calls the getMonitorables() function to get the values of all the monitorable metrics at a particular simulation time step.

2. Analyse and Plan Phases

We have provided the analysisAndPlanning() function to implement the analyse and plan phase of the MAPE-K feedback loop. In our simple example, we are selecting the topology randomly at each time step. This can be implemented using more intelligent decision-making algorithms such as Reinforcement Learning [4] and Evolutionary Computation techniques [3, 5].

3. Execute Phase

Once the topology is selected, the adaptations are performed using the Effector functions provided in Table 2. We use the setNetworkTopology() function to set the topology at a particular time step as shown in the execute() function of the MAPE_KLoop class

Run the MAPE-K loop at each Simulation Time Step

In order to run the MAPE-K loop at each simulation timestep, create an object of the MAPE_K class and execute the feedback loop by calling the run() function of the MAPE_K class at each simulation time step.

For this purpose, create a new class named *Test*. Copy and Paste the following code to it.

```
import rdm.management.Effector;
import rdm.management.NetworkManagment;
import rdm.management.Probe;
import rdm.management.RDMSimulator;

public class Test {
public static void main(String[] args) {
  //Step: 1 Load the configuration settings
NetworkManagment nm=new NetworkManagment();
//Step: 2 Instantiate the probe and effector
Probe probe=nm.getProbe();
Effector effector=nm.getEffector();

//Step 3: Instantiate the mape-K feedback loop
MAPE_KLoop loop=new MAPE_KLoop(probe,effector);

//Run simulation for the number of simulation runs defined to execute the feedback loop
for(int timestep=0;timestep<NetworkManagment.simulation_properties.getSimulationRuns();
timestep++) {
//start the feedback loop
loop.run(timestep);
}
RDMSimulator.displayResults(args);

}

}
```

4.2 Executing the program

Step: 1

In order to execute the program, first of all set the configuration parameter values in the "configuration.json" file as shown in Fig 6. You can open the file by doubling clicking on the file in the Project Explorer pane.

Step: 2

Once the configuration parameters are set, run the program by clicking on the run button[5]. It will display the graphs showing the satisfaction levels of the monitorable metrics and the results log per time step as shown in Fig 7.

A complete coded example for *TestRDM* is provided as part of the RDMSim package.

[5]Make sure the VM arguments are set

configuration.json Notepad

File Edit Format View Help

```
{
        "time_steps": 100,
        "mirror_number": 25,

        "link_threshold": 165,
        "bandwidth_threshold": 2800,
        "writing_times_threshold":1550 ,

        "topologies": ["mst", "rt"],

        "mst_active_links": [24, 175],
        "rt_active_links": [150, 250],


        "current_scenario": 0,

        "deviation_scenario_0_mst_links": [0, 0],
        "deviation_scenario_0_mst_bandwidth_consumption": [0, 0],
        "deviation_scenario_0_mst_writing_time": [0, 0],
        "deviation_scenario_0_rt_links": [0, 0],
        "deviation_scenario_0_rt_bandwidth_consumption": [0, 0],
        "deviation_scenario_0_rt_writing_time": [0, 0],


        "deviation_scenario_1_mst_links": [9, 12],
        "deviation_scenario_1_mst_bandwidth_consumption": [0, 0],
        "deviation_scenario_1_mst_writing_time": [0, 0],
        "deviation_scenario_1_rt_links": [0, 0],
        "deviation_scenario_1_rt_bandwidth_consumption": [0, 0],
        "deviation_scenario_1_rt_writing_time": [0, 0],
```

Figure 6: Configuration File

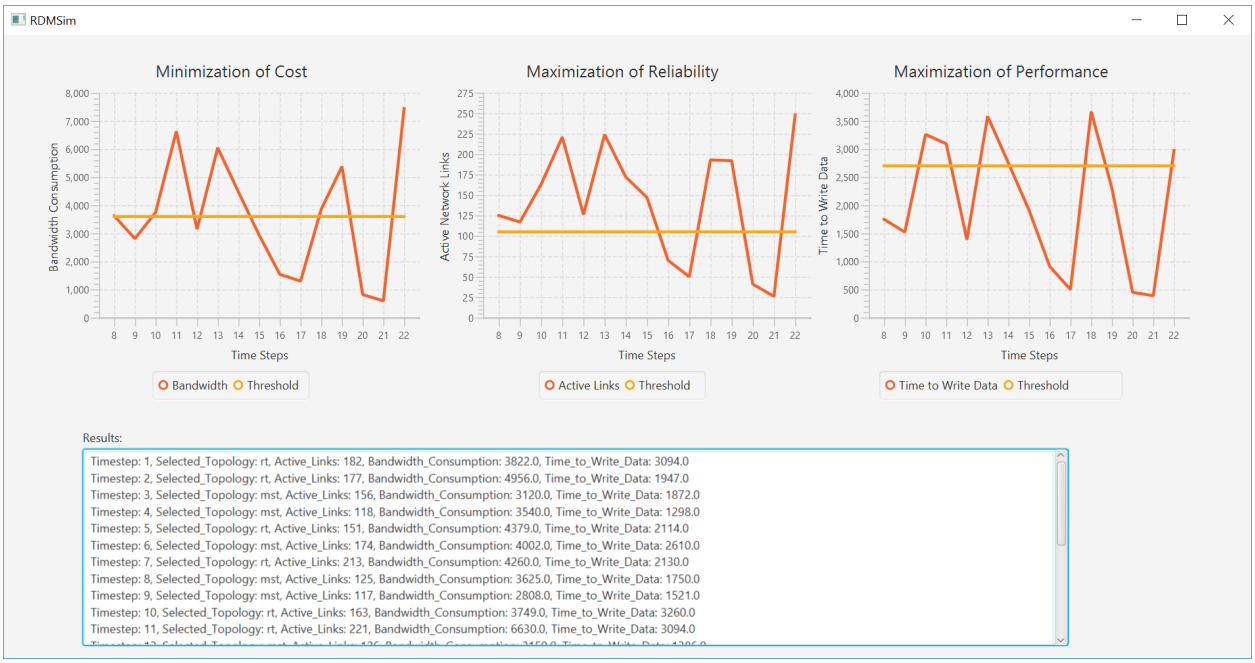

Figure 7: Graphical User Interface

References

[1] M. Ji, A. C. Veitch, J. Wilkes *et al.*, "Seneca: remote mirroring done write." in *USENIX Annual Technical Conference, General Track*, 2003, pp. 253–268.

[2] K. Keeton, C. Santos, D. Beyer, and J. Chase J.and Wilkes, "Designing for disasters," *USENIX Conference on File and Storage Technologies, Berkeley*, 2004.

[3] E. M. Fredericks, "Mitigating uncertainty at design time and run time to addressassurance for dynamically adaptive systems," *Michigan S.University. PhD Thesis.*, 2015.

[4] D. M. Roijers, P. Vamplew, S. Whiteson, and R. Dazeley, "A Survey of Multi-Objective Sequential Decision-Making," *Journal of AI Research*, vol. 48, 2013.

[5] A. Ramirez, B. Cheng, N. Bencomo, and P. Sawyer, "Relaxing claims: Coping with uncertainty while evaluating assumptions at run time," *MODELS*, 2012.