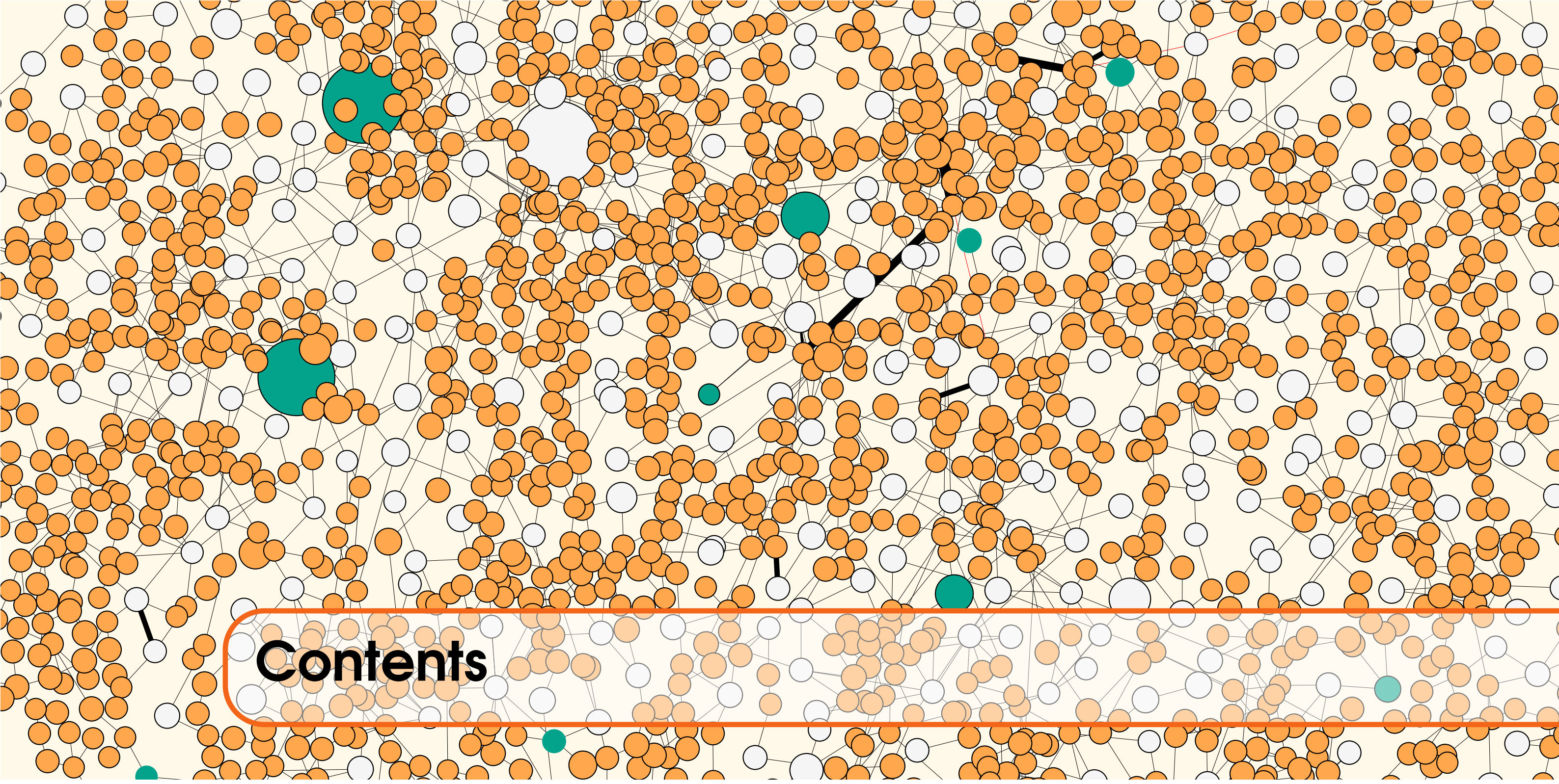
SFINA Manual

Simulation Framework for Intelligent Network Adaptations

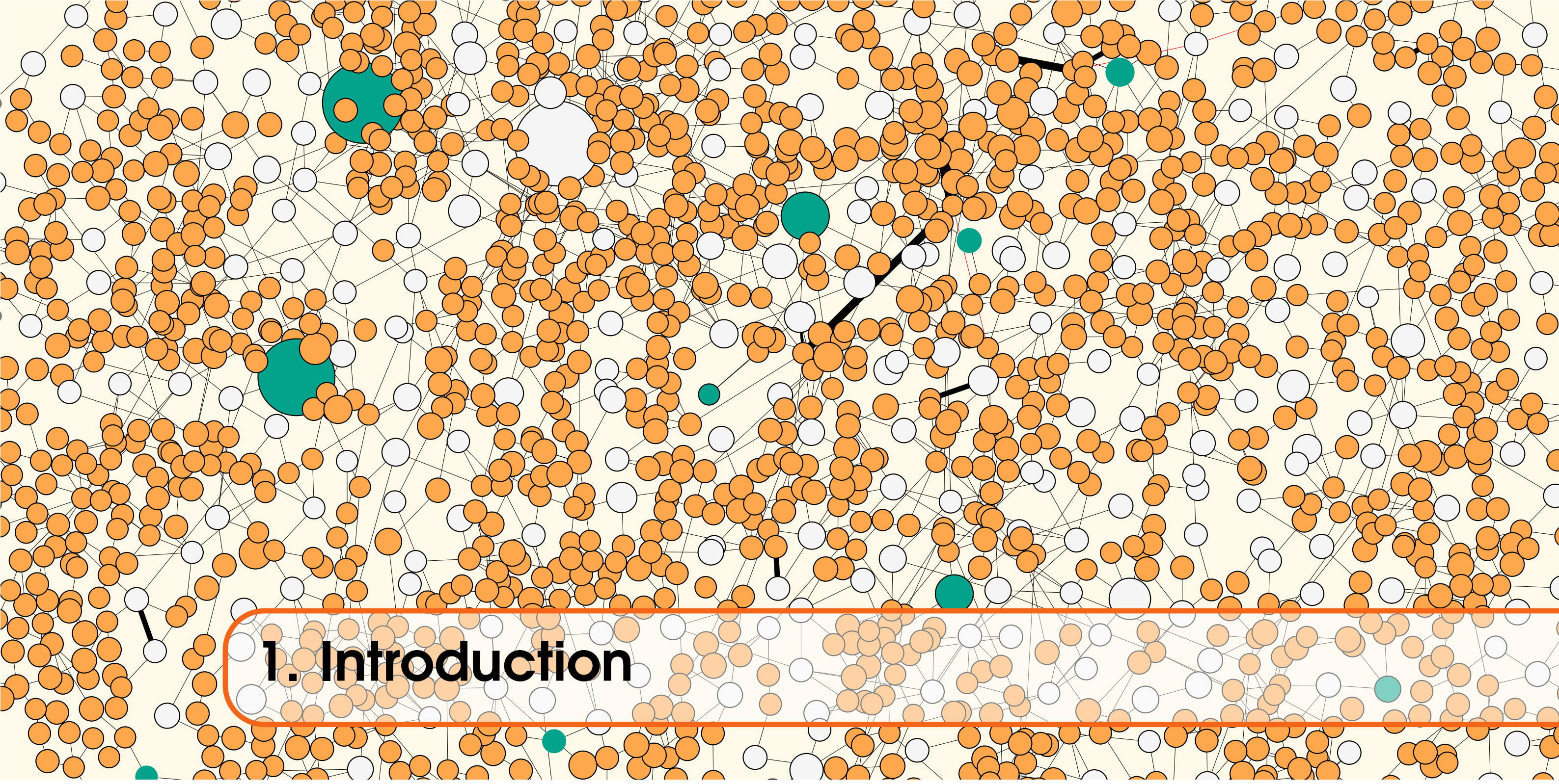
Chair of Computational Social Science, ETH Zurich



Contents

Todo list

General



1. Introduction

This manual explains the functionality of SFINA and gives hands-on advice on how to modify and extend it for one's own needs. If you implement new core functionality, we would be happy to integrate it in the official release for everyone to use. So please contact us in this case or if you have feedback for improvement.

SFINA is a simulation tool for flow networks, highly adaptable for different application domains and scenarios. A flow network is a collection of nodes between which quantities can be exchanged through links. For example in power networks, the nodes are generator/load buses connected by transmission lines through which real and reactive power can flow. Currently a fully functional power simulation is implemented, including cascading failure simulation under different attack scenarios. We are also working on extending the functionality to simulate multi-layer inter-dependent networks. For the future we envision the implementation of more domains such as information, transportation, water or gas. This will provide a unique tool to simulate several scenarios of interconnected multi-domain networks.

This simulation tool is intended to be used for researchers to study the stability, growth and adaptation of flow networks. The goal is to support decision and policy-making by providing a tool for optimization. However, by keeping it very flexible and adaptable, we hope that many others will use it for their own studies with different approaches and use-cases.

1.1 Glossary

flow network

A mathematical graph with physical properties. A collection of nodes connected by links through which (conserved or nonconserved) quantities flow.

node

A point in the network to/from which links point and which can have any number of properties, defining its behaviour.

link
A connection between two nodes, having a defined direction from a start node to an end node. Can have any number of properties, defining its behaviour.

topology
The network structure created by nodes and links.

flow
Data about the quantities flowing through the flow network, i.e. through the nodes and links.

domain
The general physical setting of the flow network, defining which quantities of the nodes and links are necessary in order to compute the flow through the network. Examples: Electrical power, transportation, information, ...

backend
Code that can compute the flow through the nodes/links if the necessary values for its domain are provided. Can converge or not converge, whose meaning has to be understood physically.

event
A change in the flow network, or its nodes/links respectively.

1. Event **feature**: Abstract notion of the network part (topology, flow, system).
2. Event **component**: Node or link.
3. Event **parameter**: Defining what is to be changed.
4. Event **value**: New value for the corresponding parameter.

agent
Implementation of SFINA functionality and its applications.

time step
Discrete time steps of the simulation, which has to be a predefined number, i.e. cannot be determined during runtime. At the beginning of a time step input data is loaded if it is provided. At the end of each time step the measurement data generated in this time step is written to a log file for later processing. A time step can contain a dynamic number of sub-simulation-steps called iterations. Synonyms: Epoch, Simulation Time.

iteration
Divides the time step into smaller entities, encapsulating the smallest possible simulation steps being measured. At the beginning of each iteration events in the queue for the current time step are executed. At the end of each iteration measurement metrics are saved and the network data at the current state is written to the output files. The number of iterations is determined during runtime: If there are pending events scheduled for the current time step (e.g. generated during the current iteration), then another iteration is triggered.

measurements
The functionality to save measurement data to a log file for later processing. Measurements are implemented in the BenchmarkSimulationAgent in the Flow Monitor repository and save several important metrics for each iteration and time step. At the end of each time step, they are written to a log file in the peerlets-log directory. This log file can be loaded after the simulation for further calculations and processing. This is implemented in the BenchmarkLogReplayer class in the Flow Monitor repository, which complements the BenchmarkSimulationAgent.

1.2 Install and setup

Depending on the goal one may want to get the full source code of SFINA or only the compiled files for easy use of the current functionality.

1.2.1 Using the compiled code

This is the way to go, if the current core functionality is sufficient, meaning the currently implemented domain and backends for this domain. It is still possible to implement complex functionality for different experiments on top of this.

1. Download SFINA.jar and necessary libraries from the website.
2. Create your own project in the IDE of your choice (Eclipse, Netbeans or others).
3. Import SFINA.jar as an external library.
4. Create an experiment by following the instructions in section 1.3.2 on how to run simulations of already implemented applications or in section 6.1 on how to create your own application.

1.2.2 Using the full source code

If you want to extend the core functionality to your needs, namely adding new domains or new backends for existing domains, then follow these steps:

1. Get the sources by cloning from Github into the current folder on your computer. For this open a terminal window, navigate to the folder you want to use and type:

   ```
   git clone https://github.com/SFINA/SFINA.git
   ```

2. Import the project into an IDE (SFINA is developed on Netbeans, so using this would be the easiest to setup).
3. Add the necessary packages, which can be found in github.com/SFINA/SFINA/libraries.
4. Start to adjust it to your needs by following the instructions in the following chapters.

1.2.3 Example input case files

The file system is explained in more detail in section 2.1, however a collection of example input case files can be found in the repository www.github.com/SFINA/InputCaseCollection. They can be used in any of the following examples (of course for the corresponding domain, currently mostly power flow simulations).

1.3 Simulations and measurements

1.3.1 The graphical user interface

SFINAGUI 1.17a offers users easy to use platform to run pre-existing experiment classes, select and run different networks for specific experiment classes, edit existing networks or create new networks, edit events or backend parameters.

Creating an interactive experiment

To use the SFINAGUI interactively, the experiment class should implement the SFINAGUIExperiment interface. SFINAGUIExperiment extends the standard runnable interface with setExperimentConfigurations() and getExperimentConfigurations() methods. To use the SFINAGUI, one needs to only instantiate the SFINAGUI with two constructor arguments: an experiment class (that implements SFINAGUIExperiment) and a boolean to indicate if the experiment is for InterdependentNetworks.

```
TestSFINAGUIExperiment e = new TestSFINAGUIExperiment();
new SFINAGUI(e,false).setVisible(true);
```

Running an experiment with pre-existing networks

When SFINAGUI is run, the experiment files existing in the experiments/ folder of the root directory will show up on the side panel.

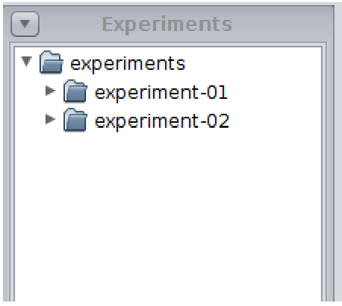


Figure 1.1: List of pre-existing experiment files on side bar.

To run the experiment with pre-existing experiment files, the user simply needs to right click on the experiment on sidebar and click on *Run Experiment* as shown in figure.

Figure 1.2: Pre-existing networks can be used ot run experiments.

Alternatively, users can also go to *Experiment -> Run*. Before running the experiment, it is advisable to ensure that the default experiment configurations are set as desired. To confirm this, one can go to *Experiment -> Configurations* as shows in 1.3.

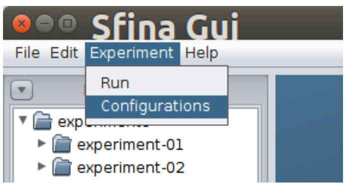


Figure 1.3: Make sure experiment configurations are as desired before running the experiment.

Upon clicking the menu item, a dialog box pops up as in Figure 1.4 which can be used to set experiment configurations.

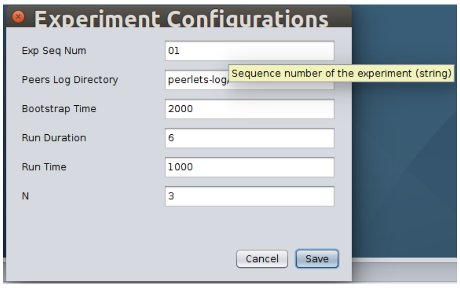


Figure 1.4: Make sure experiment configurations are as desired before running the experiment.

Once, the experiment configurations are set, users can simply go to *Experiment -> Run* to run the experiment as shown in 1.5.

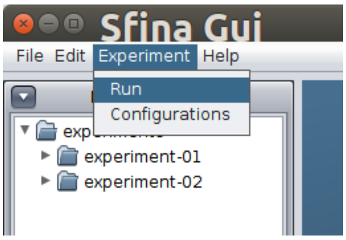


Figure 1.5: Experiments can also be run with *Run* menu item.

Creating a new network

SFINAGUI can also be used to create new networks. To create new networks click on *Edit -> Create Network* as shows in Figure 1.6.

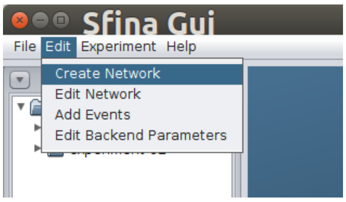


Figure 1.6: New networks can be created using SFINAGUI

This will show a dialog Figure 1.7requesting user define a list of node properties.

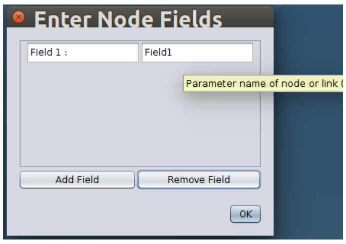


Figure 1.7: Enter the node properties

Enter the node properties and click *OK*. Another similar dialog pops up for link properties. Follow the similar steps as was done for node properties. Once done, a blank screen appears as shows in Figure 1.8on which users can interactively draw networks.

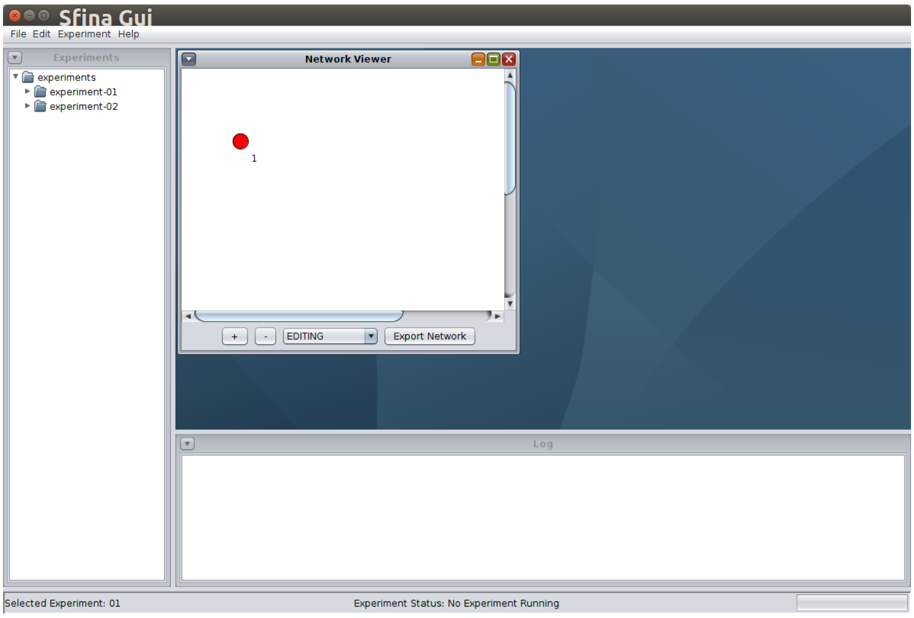


Figure 1.8: Network can be drawn interactively using SFINAGUI.

Once drawing is completed, the network files can be saved by clicking on *Export Network* Figure 1.9.

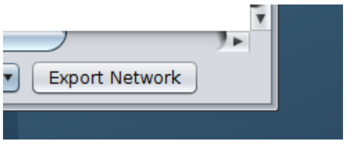


Figure 1.9: Once completed, save network in appropriate location.

While saving network files, one should make sure that the directory structure of standard SFINA framework is followed. Otherwise, SFINA framework can fail to load files and run experiments. When clicking on *Experiment* -> *Run*, directly, the default (if any) network/experiment files specified in the experiment class will be used. One can right click on specific folders on the side panel and click *Run* to run the experiment class with selected experiment/network files. Before running the experiment, one can also add events, edit backend parameters etc.

Creating Backend Parameters and Adding Events

SFINAGUI can also be used to add events or backend parameters. Note that as while running experiments using *Run* menu item, the actions *Edit Backend Parameters* and *Add Events* will be applied to currently selected experiment. To add events to currently selected experiment go to *Edit->Add Events* as shows in Figure 1.10.

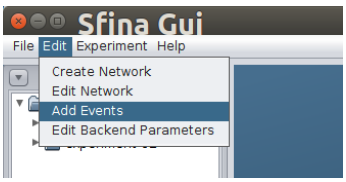


Figure 1.10: Add events to currently selected experiment .

Once the menu item is selected, user will be prompted to select the peer whose events shall be edited Figure 1.11. Upon selecting appropriate peer, users can add events to that specific peer in the

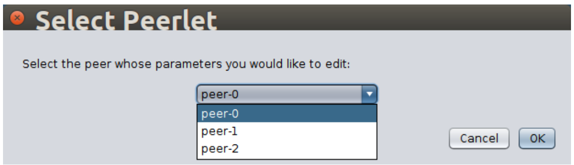


Figure 1.11: Once completed, save network in appropriate location.

experiment specified in experiment configurations Figure 1.12.

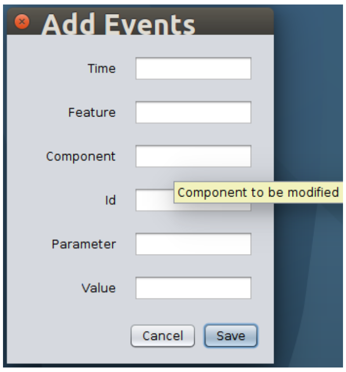


Figure 1.12: After selecting the desired peer, one can add events to the experiment.

Similarly, to edit backend parameters, users can go to *Edit->Backend Parameters* as shown in Figure 1.13.

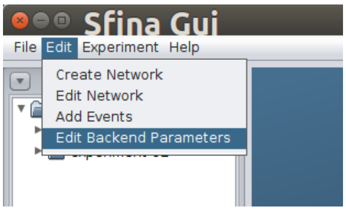


Figure 1.13: Once completed, save network in appropriate location.

Just like in add events, user will be prompted to select the peer for the experiment. Upon selecting the suitable peer, a dialog pops up which allows the user to edit backend parameters.

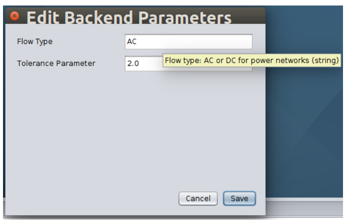


Figure 1.14: Dialog box that allows user to edit backend parameters.

Editing file system configuration

SFINAGUI can also be used to edit SFINA file system configurations. To this end, click on *File->File System Configuration* as shown in Figure 1.15.

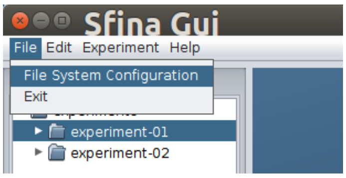


Figure 1.15: File system configurations can be edited through GUI.

A dialog that allows you to edit file system configurations pops up. When done, click on *Save* to save the modified SFINA file system configurations.

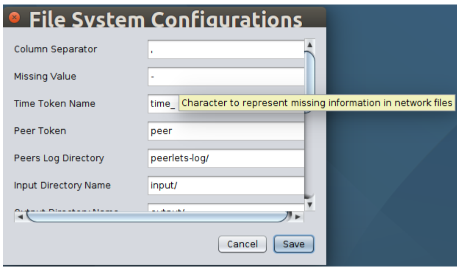
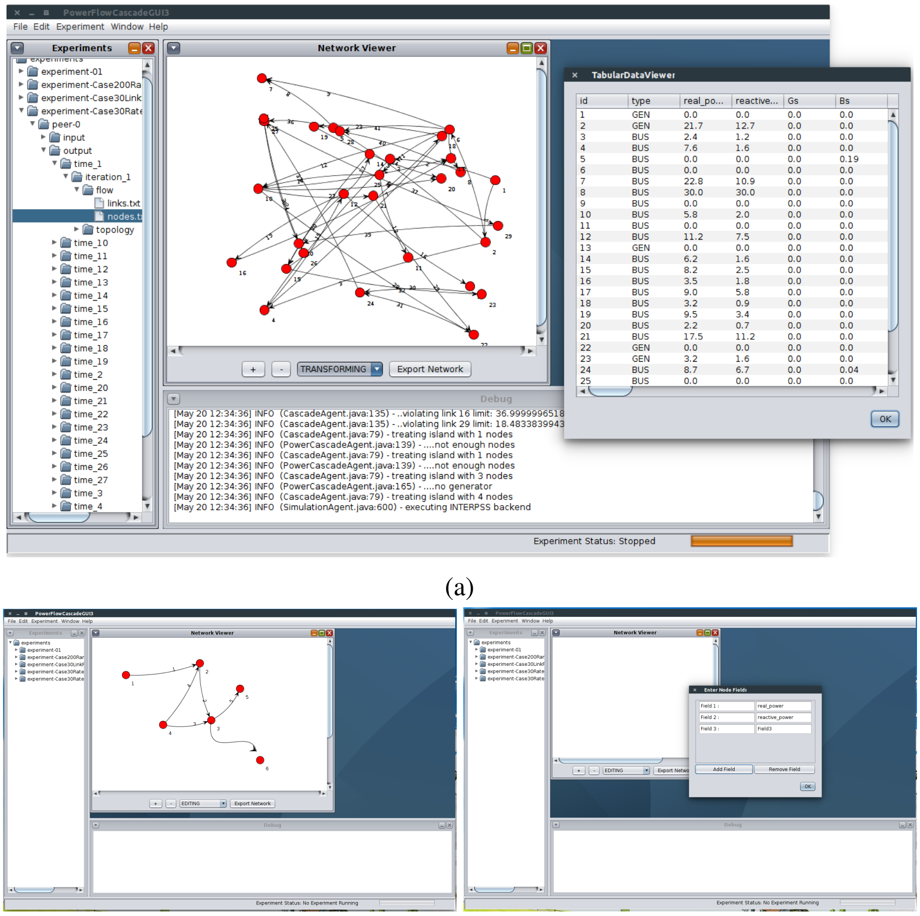


Figure 1.16: File system configurations can be edited through GUI.

Dependencies

SFINAGUI is implemented as a separate library and can be used to visually create new networks or edit existing networks. To use the SFINAGUI, one needs to include the standard SFINA libraries and dependencies and additionally Jung Graph Library, Commons IO and Collections Library. SFINAGUI makes use of Jung Graph Library and its examples to create or edit existing networks.



(a) (b) (c)

Figure 1.17: The graphical user interface (a) and how to use it to create a new network (b,c).

1.3.2 Manual setup

If the application one wants to use is already implemented and the goal is to only execute it with own parameters and input data, this section explains how to do so. In this case the algorithm and the types of performed measurements cannot be changed. But still several things can be adjusted in order to run different experiments: The name of the experiment, for how many time steps to run the simulation, the backend to use, any event to be executed at any time and when to reload a network from files or when not to do so. In order to run a simulation, one should follow these steps:

- Create a folder in experiments for this simulation, and put an input folder. Let's call our simulation "test", then we would need to create experiments/experiment-test/input
- Put all the necessary input files there
  - events.txt
  - backendParameters.txt
  - time_1/topology/nodes.txt and links.txt
  - time_1/flow/nodes.txt and links.txt
  - More time folders in the same structure as time_1, if we want to load a different new network configuration at a later time step. If no input files are provided, the simulation continues to use the data from the last simulation step. For more details on the input files and how to format them, see section 2.1.2.
- Optionally put events to be executed in events.txt. Here it is also possible to define the reloading of the first input folder by putting at time n, where this should happen:

```
n,system,-,-,reload,1
```

- Create a new class (e.g. testExperiment.java to run the PowerCascadeAgent.java application) in the following way:

```java
import application.PowerCascadeAgent;
import applications.BenchmarkLogReplayer;
import protopeer.Experiment;
import protopeer.Peer;
import protopeer.PeerFactory;
import protopeer.SimulatedExperiment;
import protopeer.util.quantities.Time;

public class testExperiment extends SimulatedExperiment{

    // define a name for this experiment
    private final static String expName="test";
    private static String experimentID="experiment-"+expName;

    // time steps including bootstrap time
    private final static int runDuration=10;

    // number of networks
    private final static int N=1;

    // other simulation Parameters
    private final static int bootstrapTime=2000;
    private final static int runTime=1000;

    public static void main(String[] args) {
        Experiment.initEnvironment();
        testExperiment test = new testExperiment();
        test.init();

        // create the instance of PowerCascadeAgent,
        // contained in a Protopeer peer
        PeerFactory peerFactory=new PeerFactory() {
          public Peer createPeer(int peerIndex, Experiment experiment){
                Peer newPeer = new Peer(peerIndex);
                // add the simulation agent
                newPeer.addPeerlet(new PowerCascadeAgent(experimentID));
                // add the agent managing the simulation steps
                newPeer.addPeerlet(new TimeSteppingAgent(
                    Time.inMilliseconds(bootstrapTime),
                    Time.inMilliseconds(runTime)));
                // add the backend,
                // in this case Power Simulation with InterPSS
                newPeer.addPeerlet(new InterpssFlowDomainAgent());
                return newPeer;
            }
        };
        test.initPeers(0,N,peerFactory);
        test.startPeers(0,N);

        // run the simulation
        test.runSimulation(Time.inSeconds(runDuration));
```

```
        // analyze measurements
        BenchmarkLogReplayer replayer =
        new BenchmarkLogReplayer(expName, 0, 1000);
}
```

First notice, that our testExperiment class extends SimulatedExperiment, which is a Protopeer class (section 2.5) providing useful functionality like time and measurements. Then it is necessary to assign to experimentID the same name as the input directory is called, in order for the experiment to find the files. The runDuration variable defines for how many time steps the simulation will run, including the bootstrap time, which is used to initialize the experiment. Defining the bootstrapTime as 2000 ms and the runTime as 1000 ms, a runDuration of 10 corresponds to 8 simulation steps. So in general: *runDuration = bootstrapTime/runTime + number of actual simulation steps*.
Different agents, encapsulating separate functionality, have to be added. For more details see section 3. Different configurations are possible, however at least three agents are necessary, one from each of the following categories:

1. A Simulation Agent, which is extended to different functionality, for example BenchmarkSimulationAgent for measurements or the PowerCascadeAgent for running cascade simulations in power networks.
2. Time Agent, for a simulation on one single network this is the TimeSteppingAgent as in the above example. For interdependent simulations between multiple networks the SimpleCommunicationAgent should be added, as explained in more detail in section 3.
3. A Domain Agent, for example InterpssBackend or MatpowerBackend (requiring a Matlab installation) in the case of a power network simulation.

The only thing left to do, is to initialize the peer we just set up, and finally executing it with test.runSimulation(...).

1.3.3 Measurements

All the measurement results are saved to a binary file in peerlets-log/experimentID/peer-0, from where it can be loaded after the simulation finished to compute, display and output measurement results. This functinality is provided by the SFINA Flow Monitor package, which one the one hand logs (BenchmarkSimulationAgent) and processes the logged information (BenchmarkLogReplayer). The BenchmarkLogReplayer shows the processed results in a table in the console and log files, and also writes the values to files in the folder results/experimentID for further processing.
To see what is going on during the simulation, logging is useful. For this first it is necessary to make sure in conf/log4j.properties the line “log4j.rootLogger=info, I, stdout” isn’t commented out. This will show information during the simulation and also the measurement output in the console and in the file log/info.log. To get a finer grained output, uncomment “log4j.rootLogger=debug, D, stdout” in the same .properties file.
All the measurement classes are in www.github.com/SFINA/Flow-Monitor.

1.4 Existing Applications

1.4.1 Cascading Simulations in Power Grids

This is an example of a more sophisticated application. As can be seen in figure 6.1 there are two levels:

1. Cascade Agent: Implementing a general cascade algorithm with a method that checks the links for overloads (i.e. the flow in the link exceeding its capacity). It implements (overrides) the runFlowAnalysis() method of the SimulationAgent, but also introduces a new method, namely flowConvergenceStrategy() which is an intermediary step before calling the backend to calculate the power flow in the network. This allows a very flexible implementation of

different (domain dependent) necessary adjustments. This is exactly what is implemented by the Power Cascade Agent, which brings us to the next point.

2. The Power Cascade Agent mainly implements flowConvergenceStrategy() from the Cascade Agent. It first checks if the current island just consists of one isolated node, returning directly non-convergence (i.e. blackout) in this case. Then it checks if there is a generator present in the island, because without power supply the island is also blacked out. Furthermore it tries to improve convergence, i.e. finding a stable solution for the power flowing in the network, by adjusting load and generation. Finally the Power Cascade Agent implements some more measurements, for example the number of islands and isolated nodes and the power demand which cannot be served (load loss).

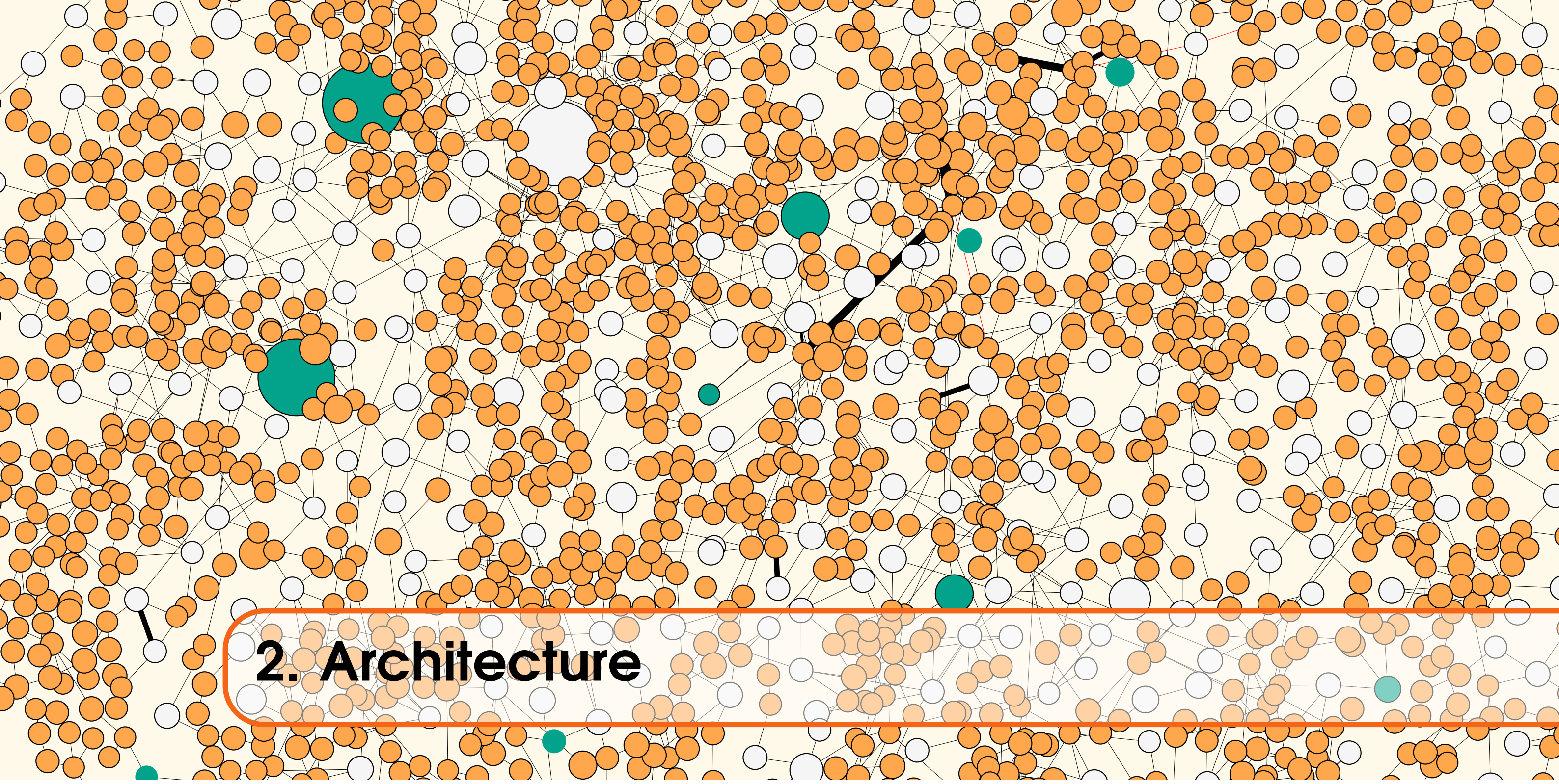
The code is in www.github.com/SFINA/Cascade.

1.4.2 Smart transformers for mitigation of cascading failures

For preventing cascading failures in power grids smart transformers can be used (Pournaras, Espejo-Uribe, *Self-Repairable Smart Grids via Online Coordination of Smart Transformers*, IEEE Transactions on Industrial Informatics, 2016). This is implemented in the PowerMitigationAgent which extends the PowerCascadeAgent and introduces these smart transformers into the network. It overrides mitigateOverload() method defined in PowerCascadeAgent (see above), where it adds the sensitivity factors of each transformer, initial flow of each link (in percent) and initial angle to some links, which are defined to be transformers by the input files. The optimization problems presented in above mentioned paper are modeled and solved with the Gurobi Java API and Gurobi 6.5.2, respectively. The code can be found in www.github.com/SFINA/Smart-Transformers.

1.4.3 Disaster spreading in complex networks

DisasterSpreadAgent is another example of application. It implements the spread of disaster in complex networks and the mitigating strategies. DisasterSpreadAgent extends the SimulationAgent. Similar to the above mentioned CascadeAgent, DisasterSpreadAgent overrides the runFlowAnalysis() method where domain specific flowAnalysis() method is called. This class also implements the various mitigation strategies. The code can be found in www.github.com/SFINA/Disaster-Spread.



2. Architecture

For running experiments which are already implemented it is only necessary to understand the file system and the concept of events. For a deeper understanding and to implement own applications, the information about the flow network, the different agents and Protopeer will come in handy.

2.1 File System

The SFINA file system allows easy input and output of all the network information needed for the simulation. Every agent has its own file system, where for each experiment a new directory has to be created and provided with input data and configuration files. The simulation automatically performs the output in the same format as the input, i.e. one folder per time step, however extended by finer grained simulation steps (iterations). The notion of time and iterations is explained in the following paragraphs. The file system is illustrated by figure 2.1, where input files or folders that show a solid contour have to be provided, the dashed ones are optional, as explained in more detail below.

2.1.1 Location of the files

Each experiment is assigned a string identifier, defined when running experiments (section 1.3.2). The input files of an experiment with the identifier “someExperiment” have to placed in the folder experiments/someExperiment, where “someExperiment” can be any string.

2.1.2 Configuration files

Two configuration files are part of an experiment, providing settings for the currently used backend (backendParameters.txt) and defining events to be automatically executed (events.txt). The backend parameter file doesn’t have to be provided (i.e. will not result in an error), however depending on the backend certain parameters might be necessary. This is for example the case of AC or DC in the power domain. The event file injects scheduled events to be executed during runtime and can be provided, but the simulation will also run without it.

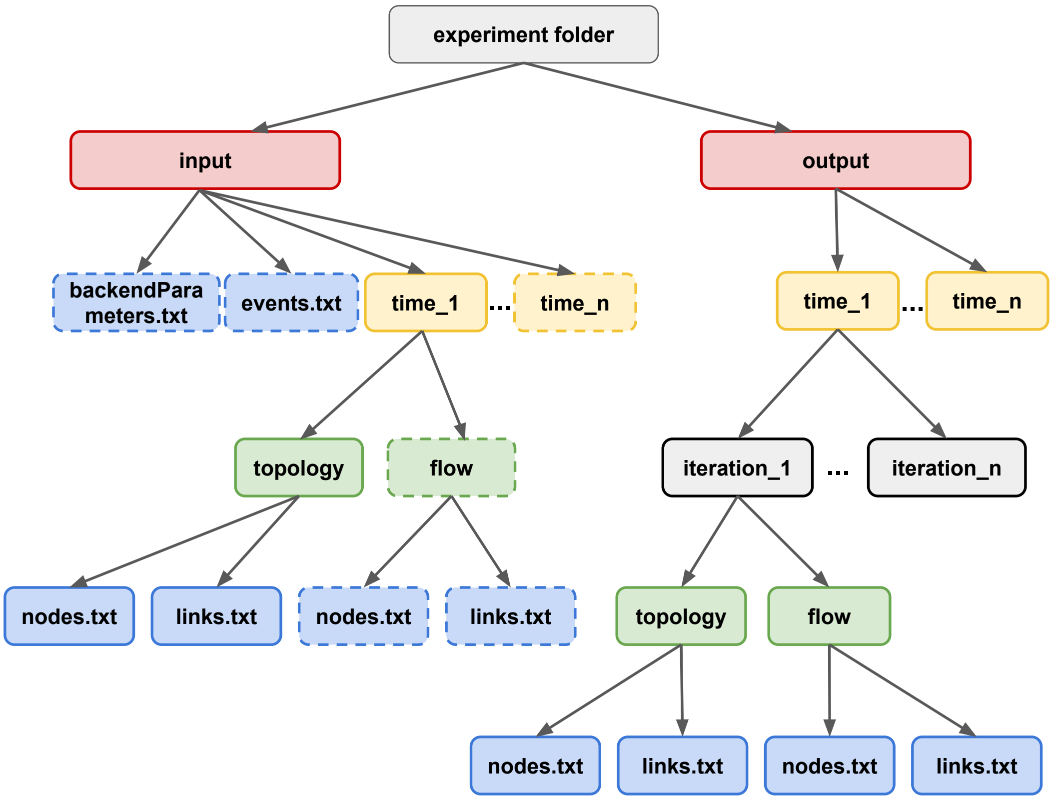


Figure 2.1: Visualisation of the SFINA file system. In input, files/folders with dashed boxes don't have to be provided. The output folder is created automatically.

The domain parameters file is specified in the format "name=value". The currently available settings are summarized in table **??**.

File	Name	Possible values
backendParameters.txt (power domain)	flowType	AC, DC
	toleranceParameter	value of type double
backendParameters.txt (disaster spread domain)	-	-

Table 2.1: Parameters that can be specified in the parameter files. Currently no backendParameters.txt for disaster_spread domain are necessary.

The events file has the following format, defining one event per line with comma separated values:

```
time ,feature ,component ,id ,parameter ,value
2,topology ,link ,10,status ,0
3,flow ,link ,9,resistance ,0.1
3,system,−,−,reload ,1
...
```

For a more detailed explanation how events work and which ones are available, see section 2.2. The events are loaded at the beginning, but each of them is executed at the time specified in the first column. It allows to define certain important events before running the experiment and without the need to modify the code.

2.1.3 Network data

Each time_n folder in input as well as in output/iteration_m contains the full network data, separated into general topological information and domain specific flow information. All the output files are generated automatically, so only the input folder has to be provided in order to run an experiment. A folder time_n in input is loaded at time step n and replaces all the network information (both topological and flow) from earlier times. At least the time_1 folder with data has to be provided in order for the experiment to have sufficient data to run. If a folder at any time step is present, it will automatically be loaded, discarding the data from before and replacing it with the new one. On the other hand it will continue with the information from the time step before if no folder for the current time is present.

Topological data

The topological data provides the nodes/links with a unique ID, which can be any string, and a status attribute which specifies if it is active or not. In the latter case the simulation treats it as if it was not present, but it can be activated during runtime for example by events. Links are directed, which can be specified by the IDs of the nodes where it starts and ends. An undirected link can be simulated by creating two links, one in each direction. The formatting of the files is shown in figure 2.2.

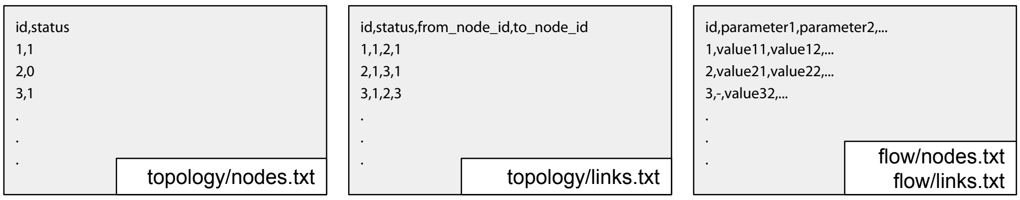


Figure 2.2: Format of topology and flow input files.

Flow data

The flow data is very flexible and has the same format for both nodes and links, as shown in figure 2.2. For each node/link information can be loaded that is needed for the flow simulation of the current domain. For example in the power domain this might be the generation output or load of a node or the resistance of a link. Similarly, for disaster_spread domain, flow data for links could be connection strength and time delay whereas for nodes this could be initial recovery rate or tolerance of the node. The values are then stored in the nodes and links of the ID specified in the first column.

The flow data is domain specific and therefore has to be defined seperately for each one. Currently this is the case for the power and disaster_spread domains, other domains are not supported yet.

In the most general case of purely topological networks, the flow data could be used to assign weights to the links or nodes. However in general it is not necessary to provide flow input files, in that case a topological network with no further attributes is generated.

Some nodes or links might be of a different kind than others, making it necessary to assign certain values to them but not to the others. In the power domain this is for example the case for nodes that generate power, which additionally need among others the power generation output information. In this case a dash (-) can be used to exclude this information from all other nodes as seen for value31 in the example above, it will be ignored during loading.

2.1.4 Manually loading network data

As explained in section 3.2 about the simulation agent, at the beginning of each time step, the corresponding input data is automatically loaded, if provided by the user. It is however also possible to initiate data loading manually, by calling the method loadData(time), where “time” is a string to be replaced with the time matching the input folder time to be loaded. It is also possible to trigger this “reload” of the time x input at time y by an event, as explained further below.

2.1.5 Changing the file system structure

The structure of the file system and names of folders outlined above is defined in a configuration file, which is loaded at bootstraping of the simulation agent (section 3.2). It is placed in the folder conf/fileSystem.conf. Besides the names of all the folders and parameter files, it also defines the column separator and the string which is used for excluding data for some of the nodes/links (missingValue) in the input data files, as explained above. These values can be changed, which is however not recommended.

2.2 Events

Events are designed for making changes to the simulation or to the network structure during runtime. They can be used in two ways:

1. Specified in events.txt to be loaded at the beginning and executed automatically at their specified time.
2. Written in the code of applications in order to change parameters “online”.

An event is defined by six parameters, specifying what action is to be executed at which time step. An overview of these parameters is given with the following table:

Parameter	Value/Description				
time	When the event is executed				
feature	topology		flow		system
component	node	link	node	link	n/a (placeholder ’-’)
id	id of the link/node whose information is changed				n/a (placeholder ’-’)
parameter	status	- status - start node id - end node id	Any flow information which was loaded or added		- reload - ...
value	0,1	- 0,1 - new node id - new node id	New value		- Time from which input data should be reloaded - ...

Table 2.2: Summary of currently available events and how to define them.

2.2.1 Using events.txt

Each line in the events configuration file defines one event, with comma separated entries for each of the six categories introduced above. As a general rule, the entries should correspond to the same strings which are also used in the other files, namely backendParameters.txt and the topology and flow data files. For example in the power domain if the goal is to change the resistance of a specific link with id 9 at time 3, it would be:

```
time , feature , component , id , parameter , value
3 , flow , link , 9 , resistance , 0.1
```

In the case of a system parameter change, component and ID are not applicable. In this case, just put a dash (-) instead. For example to reload data at time 10 form input folder time_1:

```
time , feature , component , id , parameter , value
10 , system , − , − , reload , 1
```

See section 2.1 about the file system for more details.

2.2.2 Using events in an application

The same categories are used to define events in the code, however instead of strings the appropriate Enum types are used. The constructor of the Event class takes as arguments:

```
Event(int time,
        EventType eventType,
        NetworkComponent networkComponent,
        String componentID,
        Enum parameter,
        Object value)
```

Let's look at an example, which deactivates link 20 at time step 10:

```
Event event = new Event(
        10,
        EventType.TOPOLOGY,
        NetworkComponent.LINK,
        "20",
        LinkState.STATUS,
        false);
queueEvent(event);
```

The last command schedules the events for execution at their defined time step.

2.3 Flow Network

The flow network is an object containing all the nodes and links and providing several useful methods, explained in the following.

2.3.1 Activation status and connectivity

First of all the flow network takes care of adding nodes/links to and removing them from the network and updating the network topology accordingly. The two basic topological properties a node/link can have are

1. isConnected(): A node which has (activated) links attached, or a link that has both (activated) start and end node, is connected.
2. isActivated(): A node/link that is functional, is activated. Deactivating a node, will not deactivate the attached links, but just disconnect them. Likewise, a disconnected node is still activated.

Once the network is loaded, it is possible to activate/deactivate nodes/links in the network, which will include/exclude them from any computation but will not remove them entirely. This way they can be used again later on. The nodes and links also keep track of their connectivitiy, i.e. if any other objects are connected or not. This information can be retrieved by the method isConnected(). An important method in the network is computeIslands() which extracts disconnected components from the topology of the network and returns them as an ArrayList containing a new flow network for each island. It takes all activated nodes/links into account, as can be seen in figure 2.3.

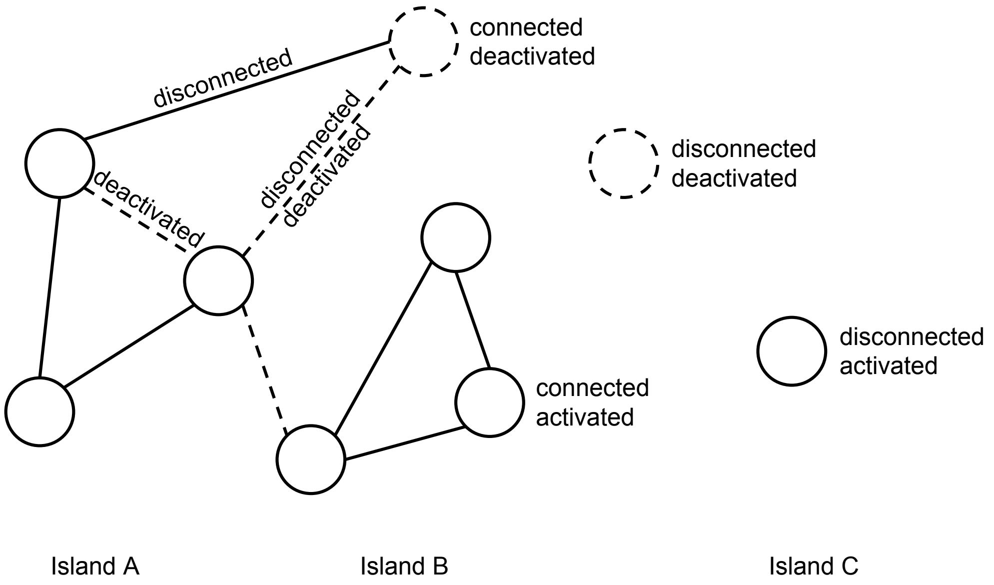


Figure 2.3: Illustration of the connectivity and activation status of nodes and links, and how they affect the computation of disconnected compontents (i.e. islands).

2.3.2 Metrics

Furthermore the flow network provides some useful methods to calculate metrics like degree distribution, clustering coefficient or to find the shortest path between two nodes. Some of these methods use the openly available JGraphT library. A more detailed list can be found in the method summary in section 7.1.

2.3.3 Flow and capacity

The flow network provides a flexible way to define the flow and capacity, meaning which physical quantities (contained in the nodes and links) is the measure of flow or how much flow a node/link can support. Use setLinkFlowType(...) and setLinkCapacityType(...) and for the nodes respectively for this purpose. This is done in the setFlowParameters() method in the simulation agent, where each domain has its default settings. When developing a new application, the developer is expected to check these. Default values are summarized in the following table.

Domain	Link flow type	Node flow type	Link capacity type	Node capacity type
Power	Real power flow from	Voltage magnitude	C rating	Max. voltage
disaster spread	...	...	...	
...				

Table 2.3: Default values for capacity and flow for the different domains.

2.3.4 Accessing the flow network and its values

Some hints which might come in handy when writing applications:

1. When writing an application which extends the SimulationAgent, the current flow network can always be retrieved by the method getFlowNetwork().
2. To add new flow information to a node or link, use the .addProperty(...) method.
3. To get or change the flow information already contained in the nodes or links, use the .getProperty(...) or .replacePropertyElement(...) method respectively. The former requires casting the return value to the correct type.

Here you see these in action, doubling the real power demand of node 15 in the current flow network:

```
FlowNetwork net = getFlowNetwork();
Node someNode = net.getNode(15); // node having ID 15
double pwr =
        (Double)someNode.getProperty(PowerNodeState.POWER_DEMAND_REAL);
someNode.replacePropertyElement(PowerNodeState.POWER_DEMAND_REAL, pwr*2);
```

A summary of methods can be found in the appendix (section 7.1).

2.4 Time and iteration

A SFINA simulation is clocked by time units, which are executed sequentially. At the beginning of each time unit new input files (toplogy, flow and events) can be provided to the simulation. During each time step several simulation iterations can be performed, depending on the logic implemented in the Time Agent. One can for example decide to perform another simulation iteration in the current time step, if the simulation did not converge in the previous iteration. By default the user triggers another iteration by calling queueEvent(event) with an event defined for the current time step. Before advancing to the next time step, the event queue is checked for remaining events and executes new iterations as long as it is not empty (for the current time).

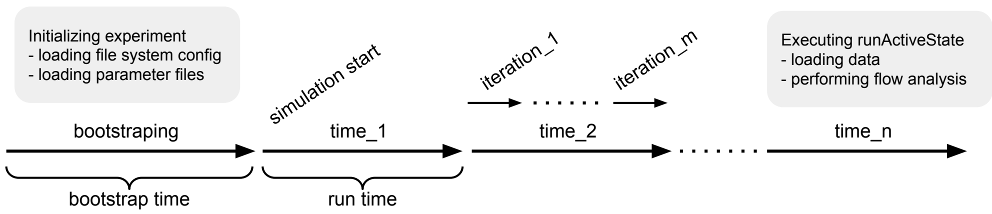
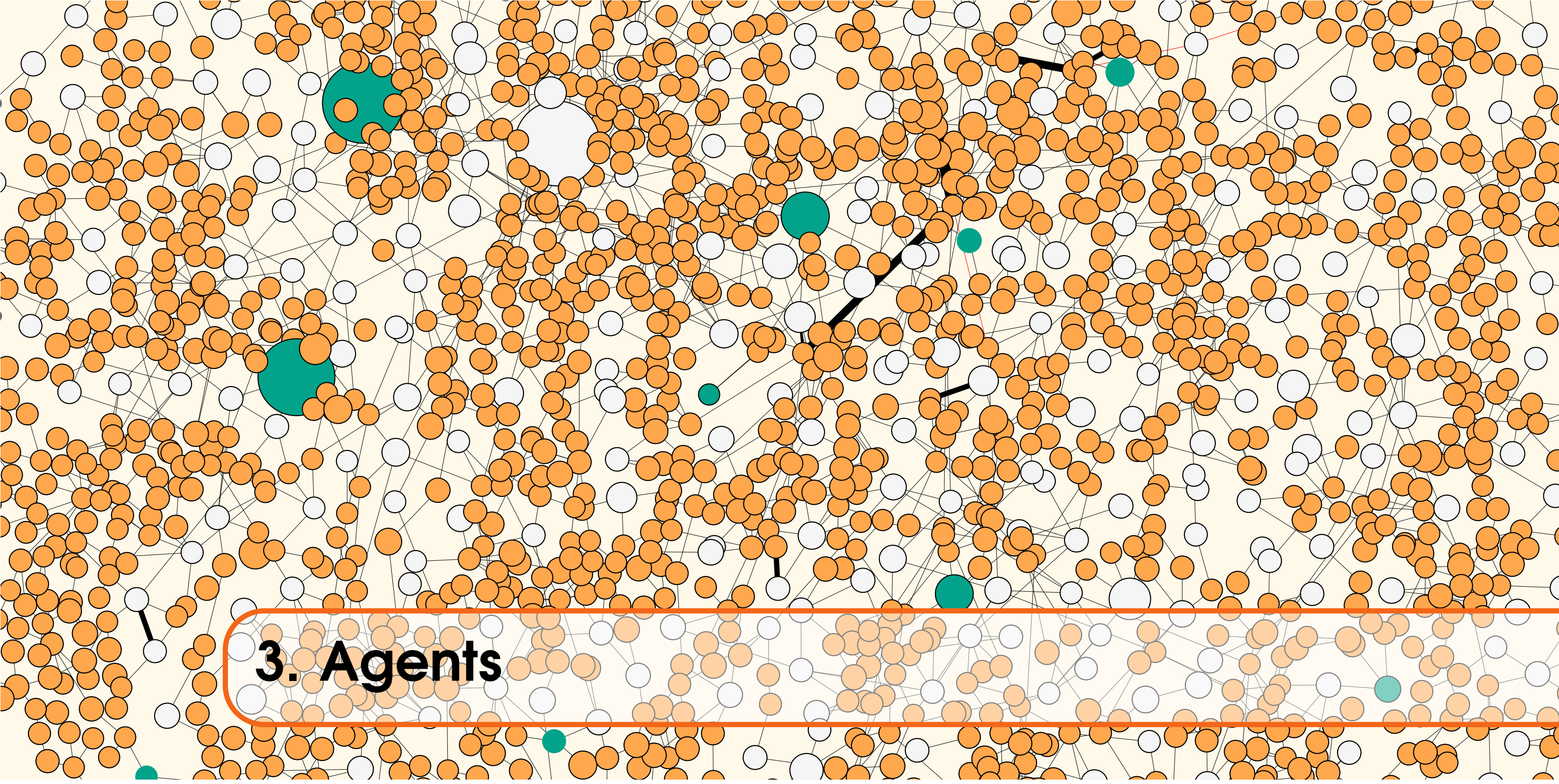


Figure 2.4: Time steps and iterations performed by the active state.

2.5 Protopeer Toolkit (redo, maybe split in part here and appendix)

SFINA makes use of the Protopeer Toolkit which was developed at EPFL university Lausanne to provide a framework for peer to peer distributed experiments (paper). It provides useful basic functionality like time based events and measurement methods. Another important concept is peers and peerlets, which allow a fully decentralized deployment. A peer is an independent simulation instance, which can communicate over the network with other peers. Every peer can contain several peerlets, which allow to further split up a peer in seperate simulations.

work over proto
section



3. Agents

The SFINA framework utilizes the protopeer library, which allows through its concepts of peers and peerlets to encapsulate different functionality separatetely. See section 2.5 for an introduction to protopeer. Each simulation is configured as a peer (see chapter 1 for an example). When creating an experiment, different agents, which implement the Peerlet interface and encapsulate separate functionality, have to be added to the peer. For example when running an experiment with interdependent networks, one can simply configure a different peer for each network. In this case one needs to add a Communication Agent and Negotiator Agent instead of the Time Agent to the peer.

The following agents have to be added to the peer as peerlets when setting up an experiment (see section 1.3.2) In brackets we note in which setup (single/ interdependent network) the agent **must** be added:

1. Simulation Agent (both)
2. Domain Agent (both)
3. Time Agent (single network)
4. Communication Agent (interdependent)
5. Negotiator Agent (interdependent)

The different agents allow a SFINA experiment to be set up in a modular way. Hence one can easily extend core functionalities of SFINA (see chapter 5) or build new applications on top of the framework (see chapter 6) through replacing some of the agents. The different agents are explained in more detail in the following.

3.1 Process flow of a basic simulation

ould be to pro-
graphic about
cess flow

In the following we present the general process flow of a simulation. This is applies to the interdependent and non-interdependent case.

1. Initialization:
   - All Peers call *init()* on their Agents (Peerlets)
   - The agents perform necessary initializations
2. Start:
   - Peers call *start()* on all their Agents.
   - *start()* should only be implemented in the Time Agent
   - Time Agent calls *runBootstrapping()* on the Simulation Agent
3. Bootstrap:
   - Simulation Agentreads necessary parameters from input files, schedules measurements etc.
   - when finished, Simulation Agent calls *agentFinishedBootstrap()* on Time Agent
4. Start simulation/ Progressing to next time step
   - Time Agent calls *progressToNextTimeStep()* on Simulation Agent
   - Simulation Agent initializes the first time step.
   - Afterwards Simulation Agent calls *agentFinishedActiveState()* on Time Agent
   - Then Time Agent progresses Simulation Agent to next iteration. In order for this to function, Simulation Agent has to signal, that it did not converge yet.
5. Progess of Simulation
   - From now on, after each simulation iteration Simulation Agent calls *agentFinishedActiveState()* on Time Agent. Time Agent then decides to
     (a) either perform another simulation iteration through calling *progressToNextIteration()* on Simulation Agent
     (b) or to skip the next iteration (makes sense for interdependent case to synchronise different networks) through calling *skipNextIteration()* on Simulation Agent
     (c) or to progress Simulation Agent to the next time step through calling *progressToNextTimeStep()*, hence return to the 4th step.

3.2 Simulation Agent

At the heart of any simulation is the simulation agent. Applications extend it and can therefore make use of its functionality (see section on adding applications for more detail). The simulation agent takes care of performing several tasks which are necessary for any simulation. These include loading and performing output, executing events and calling key methods at every time step. The time evolution of the simulation agent is illustrated by figure 2.4. Itself, it provides only barebone functionality and is extended for enabling more complex simulations, for example the

1. FlowMonitor, measuring and logging information during simulation.
2. CascadeAgent and its extention PowerCascadeAgent: Simulating cascading failures due to link and/or node overloads (i.e. capacity < flow). The latter specifically implements strategies and modifications for cascades in power networks.

The Active State

The runActiveState() method is the main runtime of the Simulation Agent. After bootstraping, during which the experiment is initialized by loading the configuration files from the experiment folder, the main simulation is orchestrated by the this method. It is executed at every time step automatically and initiates the above mentioned tasks, as depicted in detail in figure 3.1.
The Simulation Agent class is providing this minimal but extensible structure, without implementing any specific algorithm on how to perform the flow analysis at each time step. Such algorithms are

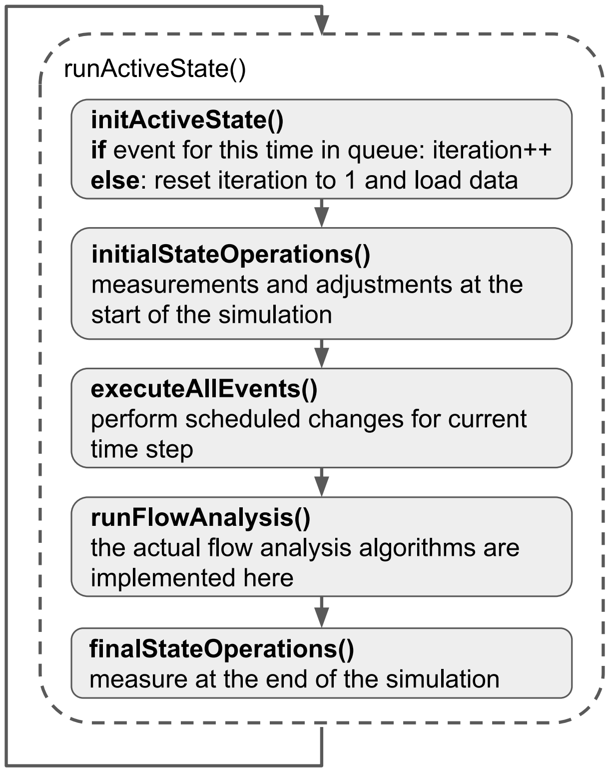


Figure 3.1: Methods executed by the active state in each time step.

meant to be implemented in applications by overriding

1. initialStateOperations()
2. runFlowAnalysis()
3. finalStateOperations().

In the SimulationAgent, the initialStateOperations and finalStateOperations methods are empty, but can be overridden in applications for more functionality. In runFlowAnalysis() callBackend() is executed to calculate the actual flow distribution in the network. This method can be called as often as necessary, i.e. is up to the developer of an application (see section 6.1 on implementing new applications for more detail).

A new iteration is initiated automatically if there are pending events for the current time step. Also the output of the files is done automatically.

The two methods initialStateOperations() and finalStateOperations() are especially useful for making measurements before and after the flow analysis. Also this allows to implement measurements, which depend on the state before and after the flow analysis. However, they can also be used to implement additional functionality, especially adjustments to the network at the beginning through events, which are then executed automatically in the executeAllEvents() method (more details in section 1.3.2 about measurements).

The callBackend() method executes the backend, which is automatically selected according to the FlowDomainAgent chosen in the experiment class. Currently implemented backends are shown in the followig table.

Domain	Backend
power	interPSS, Matpower (Matlab required)
disaster_spread	helbingetal
...	...

Table 3.1: Currently integrated backends to calculate flow in a network.

3.3 Time Agent

Organizes the time steps of the simulation, calls Simulation Agent when it should simulate the network and gets in turn notified by the Simulation Agent when it finished its iteration. Has to implement the following interface in order to function within the SFINA framework:

```
/**
* A TimeStepping Agent needs to implement this interface.
* @author mcb
*/
public interface TimeSteppingAgentInterface {

/**
* SimulationAgent can notify the TimeSteppingAgent
* that it finished its Step
*
*/
public void agentFinishedActiveState();

/**
* SimulationAgent can notify the TimeSteppingAgent
* that it finished its Bootstrap
*/
public void agentFinishedBootStrap();

/**
* Returns the current Simulation Time.
* @return
*/
public int getSimulationTime();
}
```

The following default implementation exist:

1. TimeSteppingAgent: Simulates a single network, advances Simulation Agent directly when its ready.

3.3.1 Communication Agent

Abstract class which extends TimeSteppingAgent. Thus it has the same core responsibility as Time Agent. In addition Communication Agent handles the communication between interdependent networks and is responsible for the order of execution of Simulation iterations. The following two extensions exist:

1. InterdependentCommunicationAgent: Allows for multiple networks to be connected, thus manages their synchronization and exchange of information. The execution logic is of a parallel nature: Each iteration of all networks happens in parallel, after which the networks update each other.
2. TokenCommunicationAgent: Organizes interdependent networks, but executes them in a

sequential rather than parallel order. Hence i.e. iteration 1 of network 1 happens before iteration 1 of network 2.

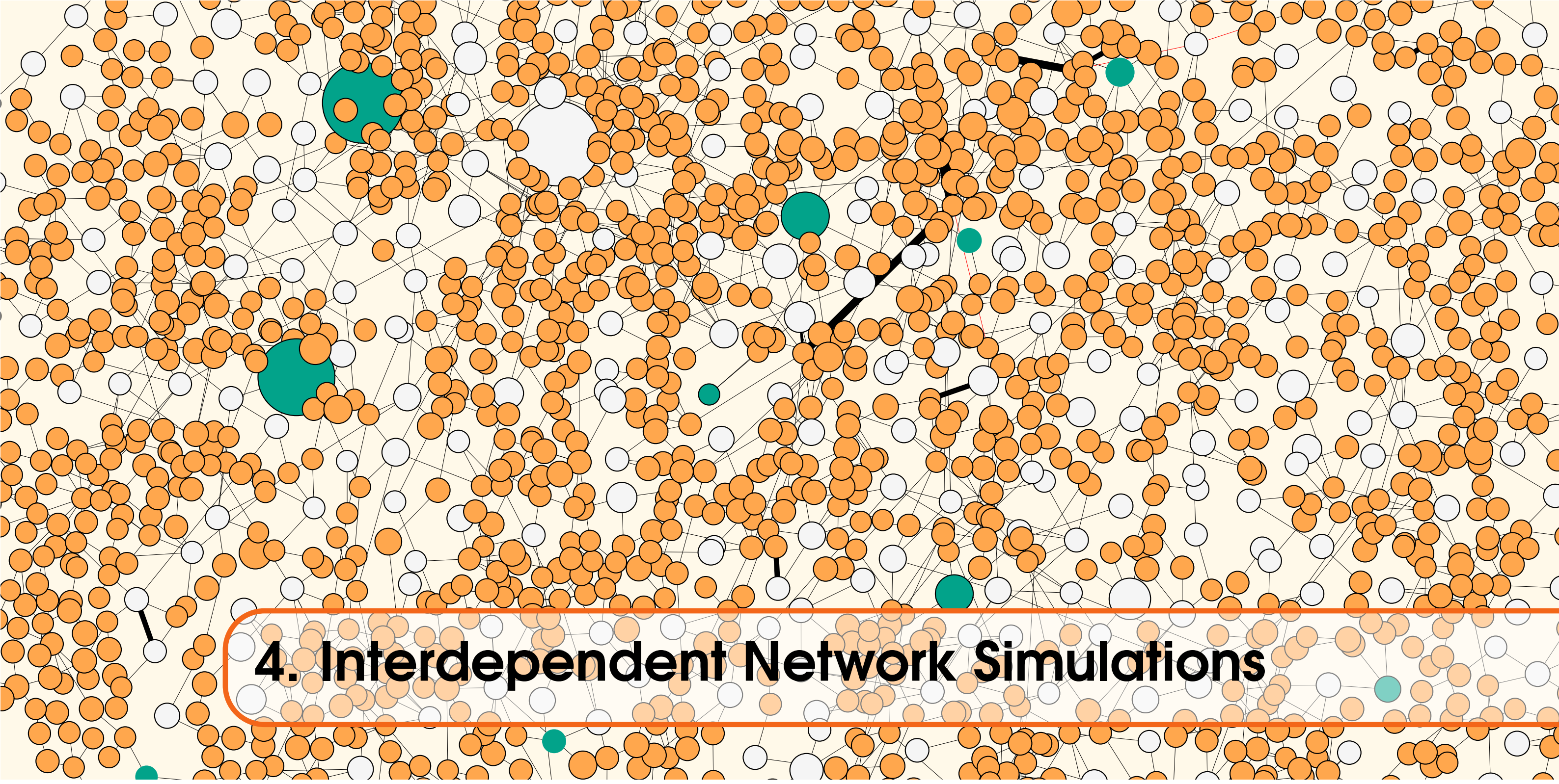
For more information on interdependent network simulations, see section 4.

3.4 Domain Agent

Provides the simulation of a distribution of flows in the network for given input data. It also handles the translation of the input file values which are specific to each domain during input and output. In the case of power simulations this can be InterpssBackend or MatpowerBackend.

3.5 Negotiator Agent

Responsible to resolve event conflicts in interdependent network simulations: If an interdependent link tries to change the same property of a specific link or node, then the negotiator decides how to reconcile them.



4. Interdependent Network Simulations

In the following we will show how to set up a basic interdependent network simulation and explain the core ingredient in detail, namely the Communication Agent.

4.1 The Communication Agent

complete communication agent section

4.2 Simulation of an interdependent experiment

Equipped with the knowledge of the setup of a single network experiment (see section TBD) and the functioning of the Communication Agent it will be easy to understand the setup of an interdependent network experiment. It namely boils down to the addition/ adjustment of the bold parts in the following experiment file. Instead of a Time Agenta Communication Agent and a Negotiator Agenthave to be added to the experiment file and the number of networks has to be specified:

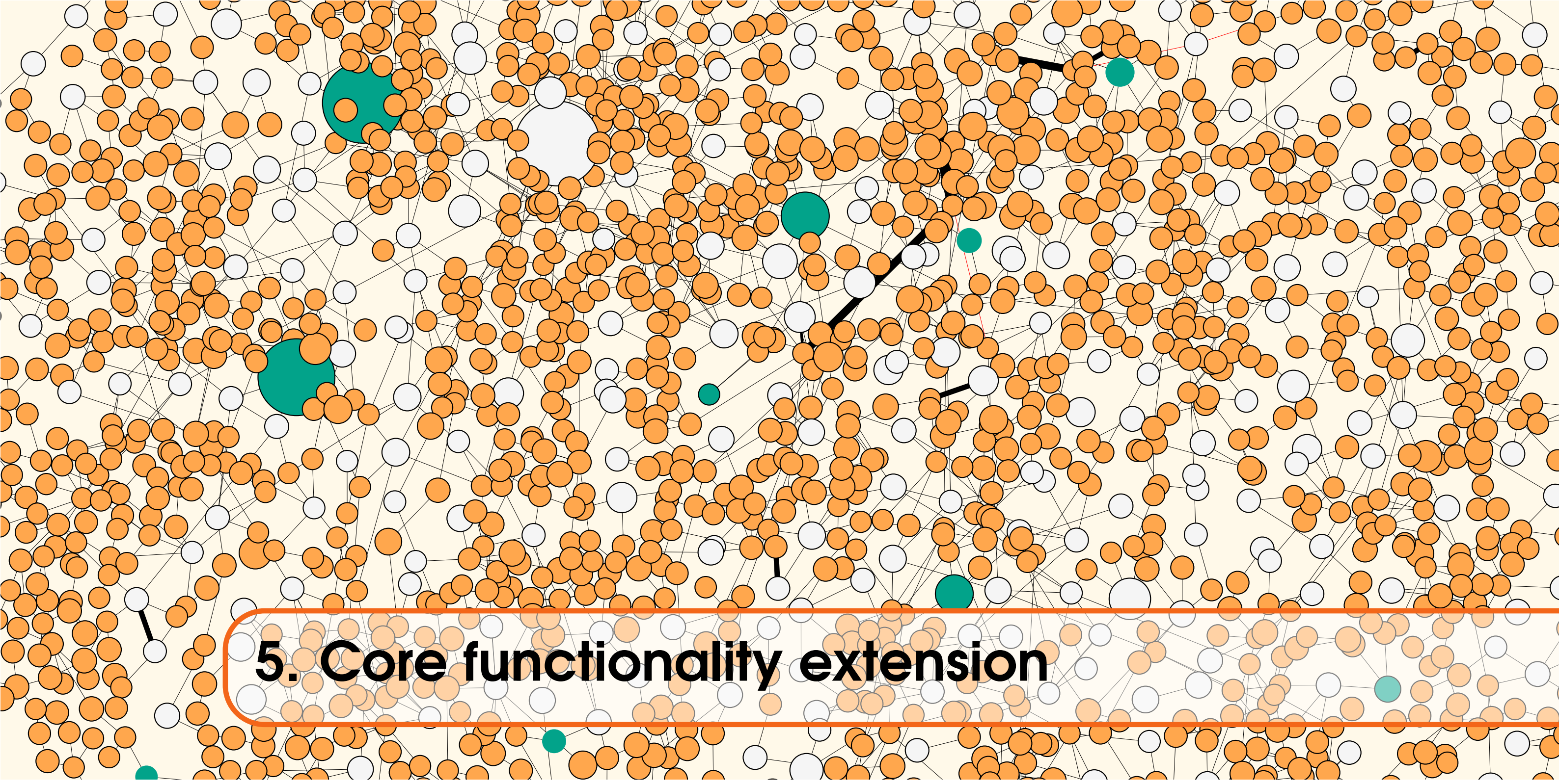
```
public class TestCommunicationAgent_communicationTimeStepping
 extends SimulatedExperiment{

        private static final Logger logger =
         Logger.getLogger(
         TestCommunicationAgent_communicationTimeStepping.class);


        private final static String expSeqNum="01";
        private static String experimentID="experiment-"+expSeqNum;

        //Simulation Parameters
        private final static int bootstrapTime=2000;
        private final static int runTime=1000;
        private final static int runDuration=6;
```

```
\textbf{        private final static int N=3;}

        public static void main(String[] args) {
                Experiment.initEnvironment();
                TestCommunicationAgent_communicationTimeStepping test =
                new TestCommunicationAgent_communicationTimeStepping();
                test.init();

                PeerFactory peerFactory=new PeerFactory() {
                        public Peer createPeer(int peerIndex,
                        Experiment experiment) {
                                Peer newPeer = new Peer(peerIndex);
                                newPeer.addPeerlet(new SimulationAgent(
                                experimentID));
                                //NECESSARY HELPER AGENTS
                        \textbf{        newPeer.addPeerlet(
                        new InterdependentCommunicationAgent(
                        Time.inMilliseconds(bootstrapTime),
                                Time.inMilliseconds(runTime),N));}
                                newPeer.addPeerlet(
                                new InterpssFlowDomainAgent());
                                newPeer.addPeerlet(
                                new PowerEventNegotiatorAgent());
                                return newPeer;
                        }
                };
                test.initPeers(0,N, peerFactory);
                test.startPeers(0,N);

                //run the simulation
                test.runSimulation(Time.inSeconds(runDuration));

        }
```

In addition to the small changes, the user needs to provide the usual topology and flow information in the same way as he or she did in the singe network case.

II

For developers



5. Core functionality extension

The way to go in order to extend the core functionality of SFINA is to replace agents with own implementations. This can be either done through extending an existing version or to implement the correct agent interface.
For adding new applications on top of the SFINA framework one has to extend the Simulation Agentor implement its interface, which is described in chapter 6.
In the following we will outline the main steps to implement own versions of the different SFINA (core) agents.

5.1 Backend Calculation: Domain Agent

A Backend is a software module that implements algorithms for the calculation of the flow in the network for given parameters. Often simulation software already exist for various domains, for example MATPOWER or InterPSS to perform power flow analysis. Three ways to implement new backends:

1. Implementing the flow calculation from scratch, using the SFINA data stored in the flow network, nodes and links. This is the most straight-forward way, and definitely the cleanest, resulting in a fully integrated backend. Depending on the complexity however, this can be quite a big task.
2. Integrate an existing backend which was written in Java. This was done for example for InterPSS. First the SFINA data has to be translated to the new backends specific format, then its power flow algorithm is called and finally the data is translated back to SFINA. It can be tedious to make sure the data is translated in the correct way, but potentially providing an easy way to integrate a new backend.
3. Integrate an existing backend written in another language. MATPOWER, which is written in Matlab, was integrated that way. The approach of 2. applies here as well, however on top an interface between Java and the other language has to be developed or integrated. In the Matlab case a package called matlabcontrol written by a third party was used to call Matlab and pass commands to it.

5.1.1 Overview of necessary adjustments

For a backend to work the following classes have to be implemented, however if only a new backend for an already implemented data structure is to be implemented the second and third steps can be omitted.

1. An extention of the abstract class *FlowDomainAgent*. This class takes care of the actual simulation. If a third-party backend is used this class translates the SFINA data to the other format, calls it and translates it back. It is then used as a peerlet in the experiment class as described in section 1.3.2.
2. An implementation of the interface *FlowNetworkDataTypesInterface*. This class properly translates the strings from the input files for the loaders. Every flow value of nodes/links (see section 2.3 for more information) is assigned to an Enum key, which should additionally be specified in a LinkState and NodeState Enum class.
3. An implementation of the *BackendParameterLoaderInterface*. This loads the domain parameters, if there are any. This should be rather straightforward. For example in the case of power simulations this loads from the file whether it is an AC or DC simulation.

The first and second step are explained in more detail in the following sections.

5.1.2 FlowDomainAgent

This abstract class always has to be extended for a new backend. It takes care of the actual flow computation in the network, given the network data. If a third-party backend is used this class translates the SFINA data to the other format, calls it and translates it back. It is then used as a peerlet in the experiment class as described in section 1.3.2.
The main method that has to be implemented is

```
@Override
public boolean flowAnalysis(FlowNetwork net){
// flow simulation goes here
}
```

It does the calculation and returns true if it found a stable solution for the flow distribution in the network (i.e. converged) and false otherwise.
Every node and link has two general fields capacity and flow, which provide an abstraction of the quantitiy flowing through the object and how much of this quantity flow it can withstand before failing. For every domain a value of the node and link flow parameters have to be designated for this purpose, which is done by the method *setFlowParameters*. The easiest is to look at an example, in this case for power flows:

```
@Override
public void setFlowParameters(FlowNetwork flowNetwork){
    flowNetwork.setLinkFlowType(PowerLinkState.POWER_FLOW_FROM_REAL);
    flowNetwork.setNodeFlowType(PowerNodeState.VOLTAGE_MAGNITUDE);
    flowNetwork.setLinkCapacityType(PowerLinkState.RATE_C);
    flowNetwork.setNodeCapacityType(PowerNodeState.VOLTAGE_MAX);
}
```

Finally there are two methods to handle domain parameters. A developer can decide if the newly implemented backend needs some user input parameters to run properly. If this is the case a BackendParameterLoader has to be implemented that loads these values into a HashMap and two methods that call this loader and extract the values from the HashMap, as in the following example for power simulations:

```
@Override
public void extractDomainParameters(){
    this.powerFlowType =
        (PowerFlowType)getDomainParameters()
        .get(PowerBackendParameter.FLOW_TYPE);
    this.toleranceParameter =
        (Double)getDomainParameters()
        .get(PowerBackendParameter.TOLERANCE_PARAMETER);
}

@Override
public void loadDomainParameters(String backendParamLocation){
    PowerBackendParameterLoader backendParameterLoader =
        new PowerBackendParameterLoader(
        this.getParameterColumnSeparator()
    );
    this.setDomainParameters(
        backendParameterLoader
        .loadBackendParameters(backendParamLocation)
    );
}
```

5.1.3 FlowNetworkDataTypesInterface

To load/write domain specific data from/to files the loaders have to be able to "understand" them. This is the purpose of the FlowNetworkDataTypesInterface. It handles the translation chain

1. Input
   (a) In the input file the strings in the first line (header) to Enum type LinkState or NodeState variables. This is done by the method *parseNode/LinkStateTypeFromString*.
   (b) For each of the following rows (each belonging to a node/link) translate the data strings to double/integer/string/etc variables. This is done by the method *parseNode/LinkValuefromString*.

2. Output, the above but in reverse
   (a) Enum type Link/NodeState → strings for the first line of the file.
   Method *castNode/LinkStateTypeToString*
   (b) double/integer/string/etc variables → strings, if necessary replacing them by the missing value string (-) if they don't apply to the data column.
   Method *castNode/LinkStateValueToString*

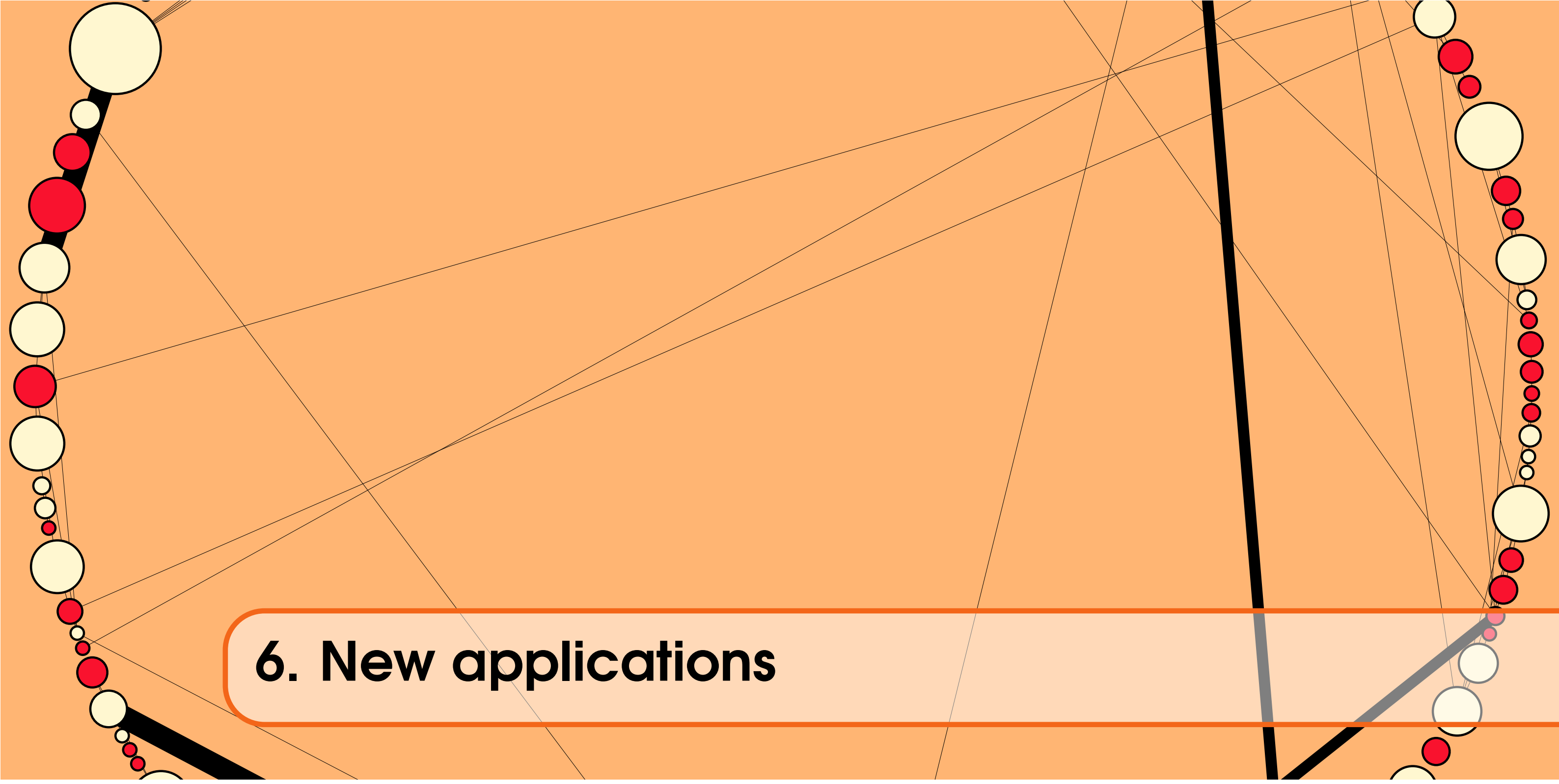
As an orientation the implementations for power simulations can be consulted in the PowerFlowNetworkDataTypes class. When implemented correctly this class should allow the data and event loaders to understand your data and load them into the nodes and links so you can use them in the for flow calculations in the new backend().

5.2 Time Stepping: Time Agent

complete time stepping agent section

5.3 Interdependent Communication: Communication Agent

complete communication agent section



6. New applications

6.1 Application architecture

SFINA is designed such that it is flexible and can be easily extended. These extensions can be the implementation of a new application as described in this section, more sophisticated measurements, and more. Currently implemented applications for showcasing the capabilities of the framework as well as for providing basic functionality, are an agent for benchmarking (i.e. making measurements), a model for cascading failures and a specific implementation of the latter for power grid simulations. New applications can be build on top of them in order to make use of their functionalities. If this is not of interest to the developer, totally new applications can be implemented on top of the SFINA core functionality. In any case, whether the SimulationAgent or one of the existing applications are used for the basis of new applications, the capability of Java to extend classes and override methods is used, which will be explained in more detail below. In figure 6.1 the current applications are shown, as well as how possible extensions could be realized.

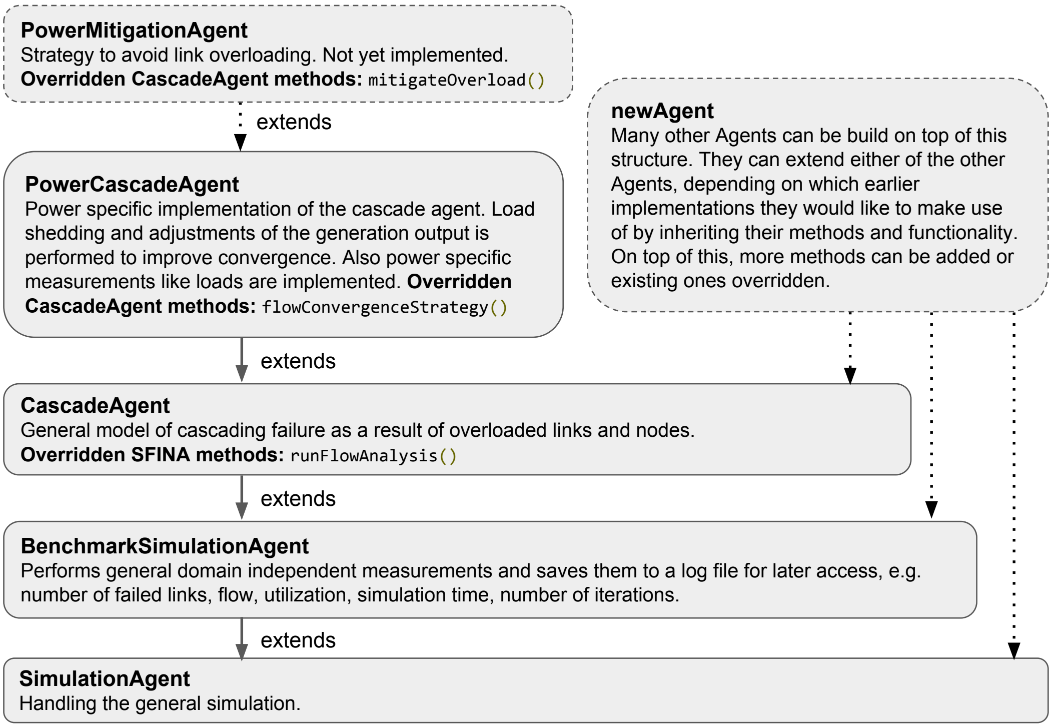


Figure 6.1: Architecture of the applications. The ones with dashed solid contours are already implemented, the dashed-contour applications are examples for possible extensions.

Graphic Needs to be replaced at some point + implement BenchmarkSimulationAgent in a modular way - Use a decorater pattern to allow for runtime additon of benchmark functionality

6.2 Implementing a new Simulation Agent

Let's have a look on how extending one of the existing agent works, taking the example of the BenchmarkSimulationAgent Its beginning looks as follows:

```
public class BenchmarkSimulationAgent extends SimulationAgent{

    private static final Logger logger =
        Logger.getLogger(BenchmarkSimulationAgent.class);

    private HashMap<Integer,
        HashMap<String,HashMap<Metrics,Object>>> temporalLinkMetrics;
    private HashMap<Integer,
        HashMap<String,HashMap<Metrics,Object>>> temporalNodeMetrics;
    private HashMap<Integer,
        HashMap<Metrics,Object>> temporalSystemMetrics;
    private long simulationStartTime;

    public BenchmarkSimulationAgent(String experimentID,
            Time bootstrapTime,
            Time runTime){
        super(experimentID,
                bootstrapTime,
```

```
                runTime);
        this.temporalLinkMetrics=new HashMap();
        this.temporalNodeMetrics=new HashMap();
        this.temporalSystemMetrics=new HashMap();
    }
    // Methods
}
```

The first thing to notice is the first line, where we define that the BenchmarkSimulationAgent extends the Simulation Agent. This makes all public methods of the latter available for our BenchmarkSimulationAgent. Then some variables are defined which are needed for this Agent to store measurement values during runtime. The constructor has to take at least the variables which the SimulationAgent's constructor needs and passes them on by using the super() method. More variables that are just used for this agent can be added here as well.
Next let's look at the methods in this agent. Most of them are just new methods necessary for the measurements, for example for initializing the measurement variables in each epoch. The important methods however are the overwritten ones, for example:

```
@Override
public void performInitialStateOperations(){
        // some methods
}

@Override
public void performFinalStateOperations(){
        // some methods
}
```

They are defined in the Simulation Agent but don't have any meaningful implementation there. By overwriting them here, we can "plug in" our own functionality, in this case making measurements before and after the actual flow simulation (see the next section on how to implement measurements for more details).
The Simulation Agent is the core of every simulation which is not supposed to be changed. Like the methods just introduced, it provides a general structure with methods that can be overwritten, namely:

1. public void performInitialStateOperations()
2. public void runFlowAnalysis()
3. public void performFinalStateOperations()
4. public void scheduleMeasurements()

To be precise, these are the only methods of the Simulation Agent that should be overridden when implementing applications. If your specific application doesn't seem to fit in this structure, then you maybe didn't try enough to simplify it, or the current implementation of the Simulation Agent is not as general as it can be. If this is the case, then you're welcome to adjust it to your needs and we would be happy to hear about your suggestions and incorporate the changes ourselves.

6.2.1 Measurements

In the last chapter we touched on the topic of measurements already by looking at how the BenchmarkSimulationAgent is implemented. Here we will explore in more detail how one could implement new measurements, either replacing or by reusing the existing ones. Two methods are designed especially to handle measurements:

1. public void performInitialStateOperations()
2. public void performFinalStateOperations()

They are executed before and after the main flow analysis respectively, and allow to save values during the current simulation, for example into one of the measurement variables temporalLinkMetrics, temporalNodeMetrics or temporalSystemMetrics. To make this more clear, let's take again the BenchmarkSimulationAgent as an example:

```
public void initMeasurementVariables(){
    HashMap<String,HashMap<Metrics,Object>> linkMetrics=new HashMap<>();
    for(Link link:this.getFlowNetwork().getLinks()){
        HashMap<Metrics,Object> metrics=new HashMap<>();
        linkMetrics.put(link.getIndex(), metrics);
    }
    this.getTemporalLinkMetrics().put(
        this.getSimulationTime(), linkMetrics);
    // initialization of other measurement variables
}

public void calculateFlow(){
    for(Link link:this.getFlowNetwork().getLinks()){
        double flow=link.getFlow();
        HashMap<Metrics,Object> metrics =
        this.getTemporalLinkMetrics().get(
                this.getSimulationTime()).get(link.getIndex());
        metrics.put(
                Metrics.LINE_FLOW, (link.isActivated()) ? flow : 0.0);
    }
}

@Override
public void performInitialStateOperations(){
    this.initMeasurementVariables();
}

@Override
public void performFinalStateOperations(){
    this.calculateFlow();
    // more measurement methods
}
```

Here you see the two overridden measurement methods introduced above, which now include new methods performing measurement tasks. The first one just initializes the needed variables to hold the measurement data. The second one gets the current flow through every line and saves it for later use. As long as these measurements are implemented in their own public methods, such as the calculateFlow() method in the above example, they can also be used by other Agents, which further extend the current one.

Measurements in SFINA are using the Protopeer framework (see section 2.5 for more details). At the end of each time step, information from the variables introduced above can be saved to a file for later use. This is done in the method scheduleMeasurements() by using log.log(int epoch, Integer iteration, Enum metric, double value), which assigns the epoch (time step) in which the measurement takes place and one or multiple tags to each stored value. The tags used are the iteration and the Metric. This information is then saved to a binary serializable file. To summarize the BenchmarkSimulationAgent logs several metrics (see table **??**) for every iteration, which is processed at the end by the BenchmarkLogReplayer, which displays and saves them.

The serializable object can be loaded after the simulation finished to perform further calculations and examination The epoch number and tags are used to retrieve the measured values. Additionally

Protopeer provides handy methods to do calculations on the data, such as statistics, calculating the mean, retrieving the maximum value, etc. An implementation of this procedure can be seen in the BenchmarkLogReplayer application.

Type	Measurement	Explanation
links	link loss	fraction of deactivated links
	link flow	average flow in the links
	link utilization	average ratio flow/capacity
	link overload	fraction of overloaded links (flow > capacity)
nodes	node loss	fraction of deactivated nodes
	node flow	avg. flow in nodes
	node utilization	avg. flow/capacity
	node overload	frac. overloaded nodes
	isolated nodes	number of nodes with no (activated) links attached
	islands	number of disconnected components
	node power loss	power demand that was reduced or can't be served
	node power loss since start	as above but since start of simulation
system	iterations	number of iterations in this time step
	simulation time	computation time of this step

Table 6.1: Currently implemented measurements in the BenchmarkSimulationAgent and BenchmarkLogReplayer in FlowMonitor package.

To summarize one has to do the following steps to implement additional measurements in an application that extends the BenchmarkSimulationAgent:

1. If necessary add your new metric (YOUR_ METRIC) to the metrics enum class.
2. Override runFinalOperations() and/or runInitialOperations() to add a method that saves your measurement values to the three HashMaps temporalLinkMetrics, temporalNodeMetrics and temporalSystemMetrics. You should add also the methods that are already there in the BenchmarkSimulationAgent in order to keep the default measurements.
3. Override logLinkMetrics(...), logNodeMetrics(...) and/or logSystemMetrics(...) to log your values to the serializable file, for example in the case of a link measurement with the method *log.log(simulationTime, iteration, Metrics.YOUR_ METRIC,*
   ((Double)linkMetrics.get(Metrics.YOUR_ METRIC)));
4. In BenchmarkLogReplayer in the methods calculateIterationResults(...) and/or calculateEpochResults(...) add a line to retrieve and compute your measurement and add it to the logger.info(String.format(...)) line and/or add another FileWriter to write it to a file for further processing like the other values.

As an example you can look at PowerCascadeAgent in the Cascade repository for such an example.

III Appendix

7. Appendix

7.1 Useful flow network methods

Names	Description
getNode("id")	Retrieve a node from the network by their id
getLink("id")	Retrieve a link from the network by their id
getNodes()	Get all nodes. Returns a collection
getLinks()	Get all links. Returns a collection
activateNode("id"), deactivateNode("id")	Activate/deactivate node by their Id
activateLink("id"), deactivateLink("id")	Activate/deactivate link by their Id
computeIslands()	Extract disconnected components. Returns ArrayList of flow networks
getShortestPath(Node a, Node b)	Computes the shortest path between two nodes
getDegreeDist()	Returns a LinkedHashMap of node degree vs number of nodes
getClustCoeff()	Compute clustering coefficient. Returns double value
getAvgNodeDegree()	Compute average node degree. Returns double value
getClosenessCentrality(Node node)	Computes node closeness centrality
getDegreeCentrality(Node node)	Compute node degree centrality

7.2 Useful nodes methods

7.3 Useful links methods

Names	Description
getIndex()	Returns the index of node
getLinks()	Returns a collection of links
isActivated()	Returns a boolean weather a node is operational or not
isConnected()	Returns a boolean weather a node is connected or not
getIncomingLinks()	Returns all the incoming links
getOutgoingLinks()	Returns all the outgoing links
getCapacity()	Returns the capacity of the node
setCapacity()	Sets the capacity of the node
addLink("id")	Adds a link specified by their id

Names	Description
getIndex()	Returns the index of link
isActivated()	Returns a boolean weather a link is operational or not
isConnected()	Returns a boolean weather a link is connected or not
getStartNode()	Returns the start node of the link
getEndNode()	Returns the end node of the link
getCapacity()	Returns the flow capacity of the link
setCapacity()	Sets the capacity of the link
getFlow()	Returns the double for flow in the link
setFlow()	Sets the flow of the link to the assigned value

Index