

PeerfactSim.KOM

The Peer-to-Peer System Simulator

-

Community Edition

Getting Started

Matthias Feldotto, Kalman Graffi
info@peerfact.org

September 19, 2013

This article is a short guide to getting started with PeerfactSim.KOM - The Peer-to-Peer System Simulator. It is designed for people who want to do evaluations on different overlay networks with use of PeerfactSim.KOM. This document helps the reader to configure the simulator, run simulations and evaluate them. Furthermore he gets the knowledge to modify parts of the overlay implementation such as to generate new statistics. This getting started document is presented with a default configuration of the chord overlay.

This guide is not a full documentation of PeerfactSim.KOM. For example, you only find a rudimentary documentation of the underlying network layer without any details of implementation. If you are interested in this part look at the whole PeerfactSim.KOM documentation at the corresponding website¹. Also, if you are interested in creating complete new overlays you should read the separate documentation for this purpose.

¹<http://www.peerfact.org>

Contents

1	The Simulator	3
1.1	The functional layers	3
1.2	The workflow of a simulation	4
2	Setup and Start	4
2.1	Download and Setup	4
2.2	Project structure	5
2.2.1	Implementation	5
2.3	Start of the simulator	5
2.3.1	Run with GUI	5
2.3.2	Run from console	7
2.3.3	Batch simulations	7
2.3.4	Evaluation of the simulation results	8
3	Configuration	9
3.1	Writing XML-Files	9
3.2	XML-Files with includes	14
4	Simulation	16
4.1	Examples for Chord	18
4.1.1	JoinOperation	18
4.1.2	LookupOperation	21
5	Overlays and Applications	23
6	Analyzer	23
6.1	Example for chord	24
7	Further Information	25

1 The Simulator

PeerfactSim.KOM is a flexible and mature event-based simulator for peer-to-peer (p2p) systems written in Java. The events follow a timeline which assures sequential processing. Events are part of so-called operations, which are generated either through the entities in the simulation or through an external actions file. Such an operation is for example a lookup in a DHT (Distributed Hash Table). Operations help to trace actions on every layer, allowing the protocols to store local state information easily and to react on operation timeouts. Operations on various layers are decoupled which allows for the combined simulation and evaluation of various protocols in parallel.

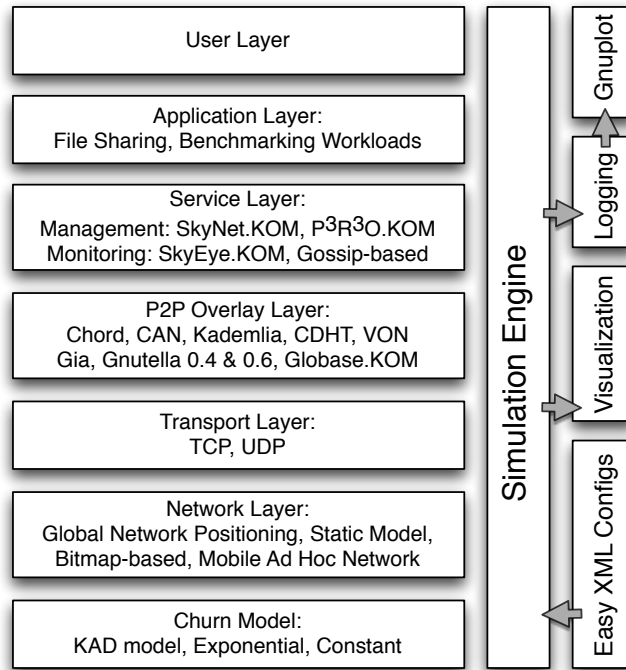


Figure 1: Overview on PeerfactSim.KOM

1.1 The functional layers

The user layer can be used to define strategies of various user types that are performed on the application layer. The application layer defines the application logic and its characteristics, such as file sharing with typical Zipf-distributed request patterns. Various advanced protocols are covered in the service layer. Management and control mechanisms for example use system aggregation monitors to constantly optimize the p2p system configuration. Such services are neither part of the application nor the p2p overlay. The service layer functions use the p2p overlay interfaces in order to provide general functional offers that improve the quality of the p2p application or create a reusable functional building block for various p2p applications. The p2p overlay layer covers structured (Chord, Re-Chord, Kademlia, Pastry, CAN), unstructured (GIA, Gnutella 0.4, Gnutella 0.6, Napster) and information

dissemination (VON, pSense, Mercury) p2p overlays with corresponding interfaces (e.g. the Key-based Routing API). The transport layer serializes messages and offers TCP and UDP as implemented protocols, which can be used in combination with the network layer to obtain realistic values for throughput, delay, jitter, loss and peer positioning. The network layer implements besides static and simple network models also advanced models, like Global Network Positioning (GNP) based on measurements from the PingER project. This network model allows to run simulations based on realistic modules based on measurements. The churn models (simulation of the temporal absence of hosts) that can be activated for time intervals are either based on measurements (in KAD) or implement popular churn behavior (exponential). In addition, there are isolation models which separate parts of the network for certain time intervals.

The simulations are conducted by the simulation event queue, which manages and schedules events in the simulation. Every event is processed at its scheduled time and logged for further analysis. The logging is twofold. First, a history of relevant events is stored for a later visualization. Second, a layer-wise protocol of the events is captured by analyzers creating simulation statistic files which can be directly fed into gnuplot. Thus, the simulator helps to easily create plotted results.

1.2 The workflow of a simulation

The workflow within the simulator is very simple. An XML-based configuration file is used to specify which implementations and configurations on which layers are to be used. The configuration file further specifies an action file, which contains the operations that are to be started by specific peers at specific time intervals. Once, the configuration is completed, the user may start a GUI to choose the configuration file and to observe the simulation status. If a visualization module was chosen, the traffic and recorded statistics over the simulation time may be observed, once the simulation finished. Independent of running simulations visualized or headless, the plotable simulation statistics are automatically generated.

2 Setup and Start

2.1 Download and Setup

To run the simulator you need the JDK 1.6 and optionally Eclipse as IDE, for modifying the code. If both are installed on your system, you should download the PeerfactSim.KOM software bundle. The current version is available at the homepage of the Community Edition². Download the file and unzip it to an arbitrary folder.

Simulating p2p systems, involves a lot of node-to-node communication through the simulated Internet. In order to determine appropriate estimations for the delay between two simulated nodes, several types of network layers are offered. One of the most precise network model, the global network positioning model, is based on measurements conducted in the Internet. The resulting delay information between 100,000+ nodes is comprised in so-called network coordinate files. They are included in the folder `config/data` in the project, you

²<http://www.peerfact.org>

also find them at the original simulator's website³. As the last step you have to compile the project. The preferred way is to use Eclipse. Open Eclipse, import the project into your workspace and build it. Otherwise you can use ANT as tool, then use the included script `build.xml`. Now the simulator is ready to use.

2.2 Project structure

Before using the simulator we will look at the project structure. As you have seen, it is prepared as an Eclipse project for easy use. You find the standard folders `src` (source code), `bin` (compiled classes) and `lib` (external libraries) in the main hierarchy. The most important folders are explained in the following:

- **src**: The whole source code can be found here.
- **test**: The source code of unit tests.
- **bin**: All compiled classes are generated here.
- **lib**: Several external libraries used in the project.
- **icons**: Graphics used in the graphical user interface.
- **config**: All configuration files for setting up simulations are found in this folder. For detail information look at the chapter for configuration.
- **outputs**: In this folder all simulated results will be saved as dat-files.

2.2.1 Implementation

The implementation is strictly divided into the API (package `org.peerfact.api`) and the implementation (package `org.peerfact.impl`). The main parts of the simulation are found in the different layers on the hosts. They all follow the component design pattern (cf. Fig. 2). In addition, the abstract factory pattern is implemented to generate instances of the components. That means, that typically for instances of nodes a factory-class is available which produces a node with desired properties. With this design pattern a configuration is easy managed (cf. Sec. 3).

2.3 Start of the simulator

There are several ways to start the simulator, the three most important ones should be presented:

2.3.1 Run with GUI

Starting the simulator using a GUI is the most practical way for a first success. You can start the GUI with the script in the main folder (`runGui.bat` or `runGui.sh`) or if you use Eclipse by running the main class `GUIRunner`.

³http://peerfact.kom.e-technik.tu-darmstadt.de/fileadmin/data/measured_data/measured_data.xml.zip,
http://peerfact.kom.e-technik.tu-darmstadt.de/fileadmin/download/mod_measured_data.xml.zip

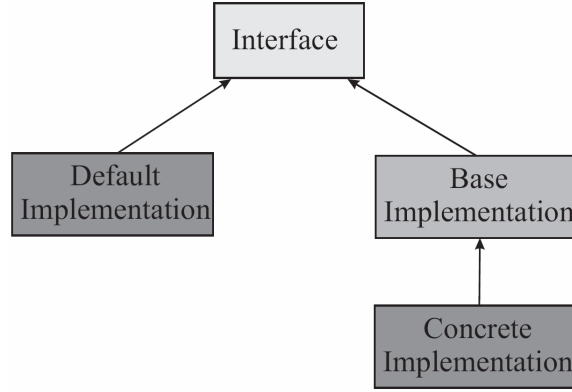


Figure 2: Components design pattern

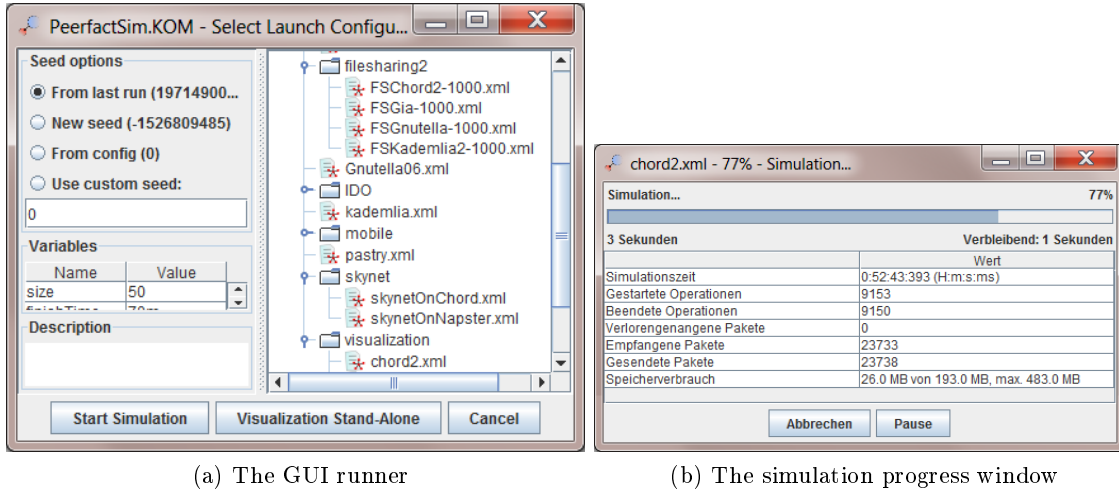


Figure 3: Configuring and running a simulation with the GUI

The GUI should appear on your screen (cf. Fig. 3a). On the right side of the window, a list of configurations is presented. You can choose a configuration file to use in the simulation (cf. Sec. 3 for details). On the left side of the window you can choose a seed for initializing the random number generator and set some variables depending on the configuration file. With the button “Start Simulation” the simulation will begin and you see the simulation progress in a new window (cf. Fig. 3b). The simulation files are editable using a text editor. In the first instance, editing the simulator configuration files is not needed.

After the simulation finishes, a visualization window opens (cf. Fig. 4) if it is configured (cf. Sec. 3). Here you can see the simulated network topology, message transfers, and node details in detail with different properties and measures. The visualization can be saved for later use.

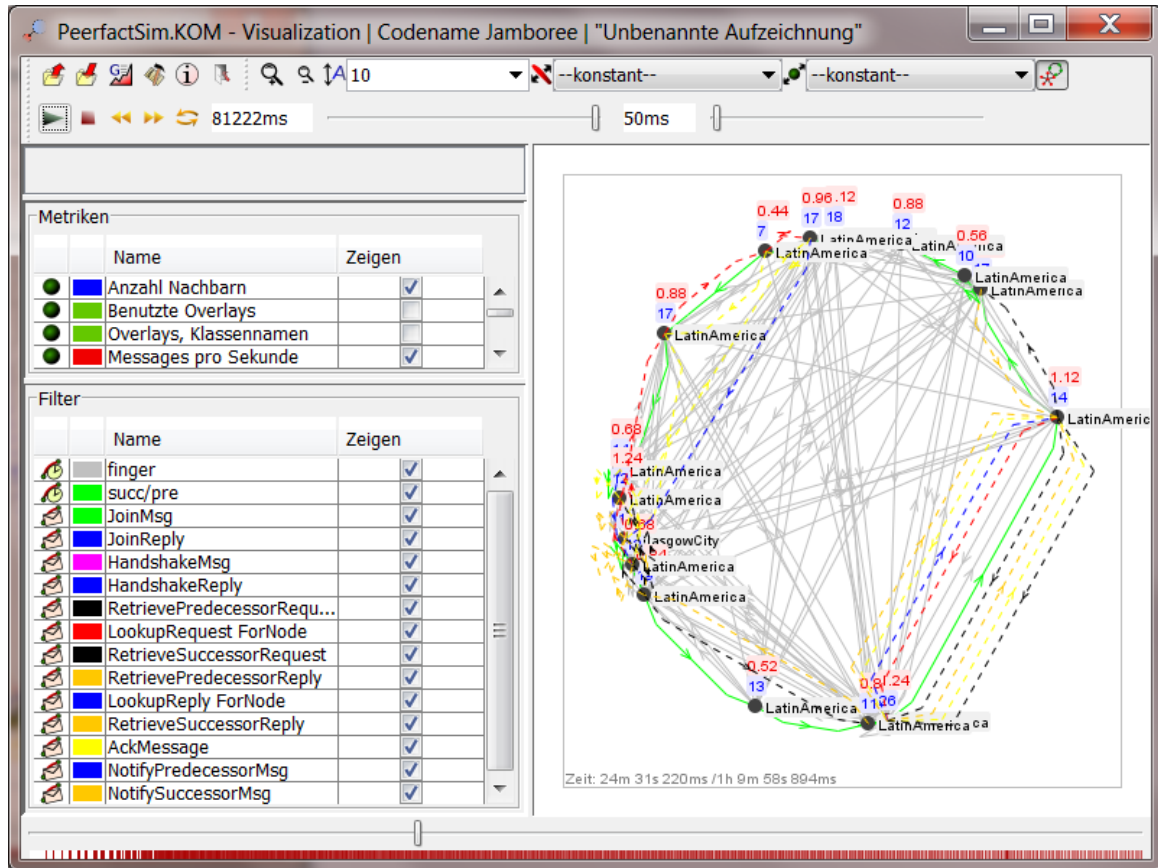


Figure 4: The visualization window

2.3.2 Run from console

You can also start the simulations with scripts. This is even advisable to use on machines without graphical user interface, for example dedicated simulation servers. To start a simulation you should use `run.bat` or `run.sh` with the relative configuration file path as first parameter. Further parameters can be used for simulation parameters (cf. Sec. 3). The follow separated with blanks and have the format `<name>=<value>`. To run directly from Eclipse use the class `SimulatorRunner`. The same simulations are conducted, as when using the GUI version.

2.3.3 Batch simulations

As third variant, you can start a complete set of simulations and configurations together with one call. This enables us to run easily comparative evaluations of peer-to-peer overlays. It is set on top of the `SimulatorRunner` and reuses it for the single simulations. In contrast, here we allow to give the variables multiple values. Together with a configuration file and a number of seeds, the runner creates the combination of all variable combinations together with the number of seeds. The `BatchRunner` allows two different modes: On the one hand a script can be generated which includes commands to start all needed simulations

and also the merging process. This script can be used on a cluster, on separate machines or the single commands can be exported to different formats. So it is very flexible, independent of the underlying system architecture for big evaluations. On the other hand the **BatchRunner** offers the possibility for the whole simulation in one call. Instead of writing a script with commands, all simulations are executed directly by the runner. Through a further parameter, the number of parallel executions in independent processes can be defined. If a simulation fails, the runner automatically restarts this single failed simulation. In the end it automatically calls the merging process.

To start a batch simulation you should use `runBatch.bat`, `runBatch.sh` or the class **BatchRunner**. The first parameter is the number of parallel executions for the second mode. If you only want to create scripts, use the number 0 as first parameter. With the second parameter you define the number of seeds used for each configuration. It should be at least 20 for acceptable statistical results. The third parameter is the relative configuration file path as in the runners before. Further parameters can be used for simulation parameters (cf. Sec. 3). In contrast to the console mode, the parameters can have multiple value. You have to specify them in the following format: `<name>=<value1>,<value2>,<value3>`.

The merging process completes the automatic evaluation. After all simulations have finished, it merges the output data and creates the single run, multiple-seed run and comparative graphics.

2.3.4 Evaluation of the simulation results

After the simulation ends, you find statistics on the simulation and the final simulation results in the **outputs** folder. To have an automatic process also in this part of the evaluation, an intelligent monitoring process was developed.

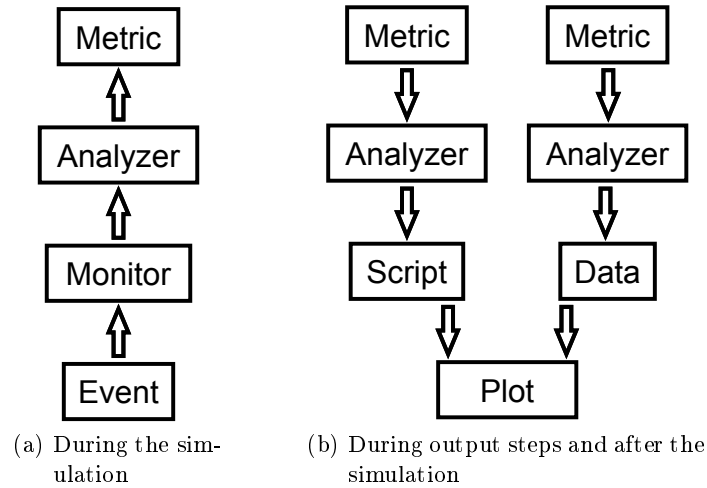


Figure 5: The monitoring process

During the simulation (cf. Fig. 5a), different events (for example the sending of a message) are called in the monitor of the simulation. This monitor forwards the event to all registered analyzers of the event type. The analyzers evaluate the event and save the needed

information in different metrics. The metrics store the overall values and additionally values grouped by the peers. After the simulation or during output steps (cf. Fig. 5b), the data of the metrics are passed to data files through the analyzers. Furthermore, the analyzers produce gnuplot scripts fitting exactly to the metrics. In the end, if you have add your gnuplot binary folder to the path variable of your operating system, the plots are produced with an installed gnuplot binary from the script and data files. As output we have three different types (for the data files as well as for the scripts and graphics):

- output of all peers ordered by the time
- output grouped by the single peers
- output grouped by the single peers and sorted by their values

Each analyzer produces these three types of data and script files once, and depending on the included metrics, different plot outputs. All plots are produced as PNG and as monochrome PDF for the direct include in publications.

If we have run batch simulations, we have a hierarchy in the output folder which contains all important files in three different layers:

1. Comparative evaluation: plot scripts and generated graphics for comparison with all parameter combinations
2. Multiple-seed evaluation: data files, plot scripts and generated graphics for one parameter combination with all merged seeds
3. Single evaluation: data files, plot scripts and generated graphics for one configuration with one seed

By having the data files, the gnuplot scripts and the graphics in the folders, the user can modify them easily for different purposes, for example for publishing in a paper. The resulting data files and also the plots depend on the used analyzers.

3 Configuration

In order to setup a simulation, a wide set of possible protocols and parameters may be chosen. All this setup information is handled in configuration files. The simulation is configured by XML-files stored in the `config`-subfolder. There are two possibilities to write a configuration file. The first one is to write every line of the file for yourself and the second one is to use predefined XML files from the `config/includes`-subfolder and include them in your configuration. We will explain both ways in this section to give the reader a overview how to write good configuration files.

3.1 Writing XML-Files

We will explain the configuration by looking at an example XML-File which is included in the simulator in `config/visualization/chord.xml`.

The first important part of the configuration file is the default section (cf. Listing 1). The variables defined here are used in the whole configuration file and their values can be used in the config file (by preceeding the variable name with a dollar sign) and can be modified in the GUI before starting a simulation.

```

1  <Default>
2    <Variable name="seed" value="0" />
3    <Variable name="finishTime" value="60m" />
4    <Variable name="actions"
5      value="config/visualization/chord-actions.dat" />
6    <Variable name="gnpDataFile"
7      value="config/data/mod_measured_data.xml" />
8    <Variable name="churn" value="true" />
9  </Default>

```

Listing 1: The default section of the configuration file `chord.xml`

The `seed` variable is used for generation of random numbers. If you choose the same seed during your simulations, you will end up with the same sequence of random numbers. The overall simulation time is defined in the `finishTime` variable, here it is 120 minutes. After this time the events are no further processed and the simulation is evaluated. In order to define the initial action of the nodes and specific action intervals, an `actions` file is defined. It defines all actions that are to be executed during the simulation. The next two variables setup the network. The `gnpDataFile` variable defines the location of the global network positioning measurements file, which is used to derive precise transmission delays between two nodes. The `churn` variable switches churn on and off.

```

1  <SimulatorCore class="org.peerfact.impl.simengine.Simulator"
2    static="getInstance" seed="$seed" finishAt="$finishTime">
3  </SimulatorCore>

```

Listing 2: The used simulator of the configuration file `chord.xml`

The next part, the simulation core is defined, `SimulatorCore`, which provides a simulation engine (cf. Listing 2). The simulation engine is in charge of queueing the events and process them in order. Beside the full qualified class name of the simulation engine there is the possibility to add parameters. Each attribute beside `class` and `static` will be handled as there exists a Setter-method for it, otherwise an exception will be thrown while initializing. To add new parameters, you add an attribute, for example `test`, with a value, for example 10 in the tag. Furthermore in the Simulator class you have to add a method `setTest(int a)`. The configurator will automatically call the method with the specified parameter while initializing.

Having setup up the simulator engine and the rough simulation times, next it is time to define the components to be used in the simulation (cf. Listing 3). One important component, although not for the final p2p protocol testing, is the network layer. As mentioned before, the network layer is in charge to determine delay times, loss rates, jitter, bandwidth and further network related aspects for each node-pair. Thus it has a great influence on the simulation results. Equal to the syntax of adding further parameters and variables to the simulator core, also here it is possible to do so. In addition to adding parameters, there is the possibility to use subtags. Subtags will be handled first and created like other components

with their own attributes (also with use of setter methods). The good thing about them is, that you can pass specific parameters to classes inside of your components. As last step the current component will be created and the sub objects will be added to the main component as a parameter of a setter-method (named like the subtag). For the network layer the objects for `MeasurementDB`, `PacketSizing`, etc. will be created and the setter methods will be called for the attributes. After that, the main component `NetLayer` will be created and for each subtag a setter method will be called with the respective object as parameter.

```

1  <NetLayer
2      class="org.peerfact.impl.network.modular.ModularNetLayerFactory"
3      downBandwidth="122880" upBandwidth="32768" useRegionGroups="false"
4      useInOrderDelivery="false" preset="Fundamental">
5      <!-- Loads a XML-File with measurement-data for latency etc. -->
6      <MeasurementDB
7          class="org.peerfact.impl.network.modular.db.NetMeasurementDB"
8          file="$gnpDataFile" />
9      <PacketSizing
10         class="org.peerfact.impl.network.modular.st.packetSizing.
11         IPv4Header" />
12     <Fragmenting
13         class="org.peerfact.impl.network.modular.st.fragmenting.
14         IPv4Fragmenting" />
15     <TrafficControl
16         class="org.peerfact.impl.network.modular.st.trafCtrl.
17         BoundedTrafficQueue" />
18     <PLoss
19         class="org.peerfact.impl.network.modular.st.ploss.
20         PingErPacketLoss" />
21     <Latency
22         class="org.peerfact.impl.network.modular.st.latency.
23         GNPLatency" />
24     <Jitter
25         class="org.peerfact.impl.network.modular.st.jitter.
26         PingErJitter" />
27     <Positioning
28         class="org.peerfact.impl.network.modular.st.positioning.
29         GNPPositioning" />
30 </NetLayer>
31
32 <TransLayer
33     class="org.peerfact.impl.transport.DefaultTransLayerFactory" />
34
35 <Overlay
36     class="org.peerfact.impl.overlay.dht.chord.chord.components.
37     ChordNodeFactory" port="400" />
38
39 <Application
40     class="org.peerfact.impl.application.dhtlookupgenerator.
41     DHTLookupGeneratorFactory">
42     <Distribution
43         class="org.peerfact.impl.util.stats.distributions.

```

```

43     UniformDistribution"
44     min="30"
45     max="120" />
46 </Application>

```

Listing 3: The used components of the configuration file `chord.xml`

Thus, in the next listing the components, i.e. network layers, are initialized. Here you see the initialization of the network layer (NetLayer), of the transport layer (TransLayer) and of the overlay layer (Overlay). Here, the overlay layer is initialized with Chord. Now, the simulator knows which layers and implementations to initialize in order to have them usable. You can also define further network layer or overlays, as only in the next step we attach them to actual nodes.

On top an application is defined. In our cases we use the a lookup generator which periodically produces lookups in the network.

The next part of the configuration file deals with the used analyzers (cf. Listing 4). Analyzers are listeners or monitors of specific events, which they log and create statistics of. For example an overlay monitor would be informed every time a message is sent or received by the overlay implementation. It maintains statistics on the number and type of message sent and the generated traffic. Further it could count the hops needed to perform a query in the overlay. However, the analyzers also flush their statistics periodically to the disk in analyzer-specific files in order to log the happenings in the simulation over time.

```

1  <Monitor class="org.peerfact.impl.common.DefaultMonitor"
2    start="0" stop="$finishTime">
3
4    <Analyzer
5      class="org.peerfact.impl.analyzer.visualization2d.analyzer.
6      VisAnalyzer" >
7      <OverlayAdapter
8        class="org.peerfact.impl.overlay.dht.chord.chord.vis.
9        ChordAdapter"/>
10     </Analyzer>
11
12    <Analyzer
13      class="org.peerfact.impl.analyzer.DefaultAggregationAnalyzer" />
14    <Analyzer class="org.peerfact.impl.analyzer.DefaultChurnAnalyzer" />
15    <Analyzer
16      class="org.peerfact.impl.analyzer.DefaultConnectivityAnalyzer" />
17    <Analyzer
18      class="org.peerfact.impl.analyzer.DefaultDHTOverlayAnalyzer" />
19    <Analyzer
20      class="org.peerfact.impl.analyzer.DefaultKBROverlayAnalyzer" />
21    <Analyzer class="org.peerfact.impl.analyzer.DefaultNetAnalyzer" />
22    <Analyzer
23      class="org.peerfact.impl.analyzer.DefaultOperationAnalyzer" />
24    <Analyzer class="org.peerfact.impl.analyzer.DefaultTransAnalyzer" />
25
26    <Analyzer class="org.peerfact.impl.overlay.dht.chord.base.analyzer.
27    ChordLookupOperationAnalyzer" />
28    <Analyzer class="org.peerfact.impl.overlay.dht.chord.base.analyzer.

```

```

22     ChordStructureAnalyzer" />
23
24 </Monitor>

```

Listing 4: The used analyzers of the configuration file `chord.xml`

In the monitor tag you can define the implementations to use for analyzing (cf. Sec. 6). In addition, the start and stop time of monitoring can be defined. The `VisAnalyzer` is a special analyzer, as its observations are used to visualize the simulations during run time, have a look at the configs in the “visualization”-folder. The visualization is overlay specific, thus you must provide a specific adapter per overlay.

The next part of the configuration file assemble the node in the simulator (cf. Listing 5). It defines either individual hosts or groups of hosts and the protocol layers they should be equipped with. Hosts can configured individually or as groups of hosts. For each one, the previously defined components from above can be used as building blocks. In addition, an oracle can be configured for analyzers, the oracle provides all necessary information about all hosts. In the beginning, the experiment size of the simulation is defined. It marks the maximum number of hosts to be specified. Then several nodes and groups are defined with distinct names, group sizes and component equipment.

```

1  <HostBuilder
2    class="org.peerfact.impl.scenario.DefaultHostBuilder"
3    experimentSize="51">
4
5    <Host groupID="GlasgowCity">
6      <NetLayer />
7      <TransLayer />
8      <Overlay />
9      <Application />
10     <Properties enableChurn="$churn" />
11   </Host>
12
13   <Group groupID="LatinAmerica" size="20">
14     <NetLayer />
15     <TransLayer />
16     <Overlay />
17     <Application />
18     <Properties enableChurn="$churn" />
19   </Group>
20
21   <Group groupID="Germany" size="30">
22     <NetLayer />
23     <TransLayer />
24     <Overlay />
25     <Application />
26     <Properties enableChurn="$churn" />
27   </Group>
28 </HostBuilder>
29
30 <Oracle class="org.peerfact.impl.util.oracle.GlobalOracle" />

```

Listing 5: The defined hosts of the configuration file `chord.xml`

Afterwards, different generators of operations are defined (cf. Listing ??). Starting at the defined time, they periodically generate churn events (caused on a model).

```

1 <ChurnGenerator class="org.peerfact.impl.churn.DefaultChurnGenerator"
2   start="90m">
3   <!-- Churn provides different statistical models. -->
4   <ChurnModel
5     class="org.peerfact.impl.churn.model.KadChurnModel" />
6 </ChurnGenerator>

```

Listing 6: The used analyzers of the configuration file `chord.xml`

The last command of the configuration file points to a separate `actions`-file (cf. Listing 7). This command defines in specific which file (named in the variable `$actions`) should be given as input to which class (Scenario class). The scenario class “`CSVScenarioFactory`” is able to interpret an action file with a specific syntax and produce simulator events in accordance.

```

1 <Scenario class="org.peerfact.impl.scenario.CSVScenarioFactory"
2   actionsFile="$actions"
3   componentClass="org.peerfact.impl.application.dhtlookupgenerator.
4   DHTLookupGenerator"
5   additionalClasses="org.peerfact.api.overlay.JoinLeaveOverlayNode"/>

```

Listing 7: The scenario of the configuration file `chord.xml`

In the next listing, an action file is presented (cf. Listing 8). The file defines different actions for the hosts, which have been configured in the configuration file. Each line contains the host (group) name, the time of the action (if a period is given, the events are uniformly distributed), and a method with parameters of the given component class (defined in the scenario tag). For the defined component class only the method name is required. Additional classes need also the class name. The `ScenarioFactory` generates operations for the actions at the defined times (cf. Sec. 4).

```

1 #Scenario Ideal
2 GlasgowCity 1s JoinLeaveOverlayNode:join callback
3 LatinAmerica 1m-8m JoinLeaveOverlayNode:join callback
4 Germany 1m-8m JoinLeaveOverlayNode:join callback
5
6 LatinAmerica 80m startLookups

```

Listing 8: The action file `chord-actions.dat`

With the given configuration file as example, we learned how to set up the frame for a simulation, define simulation times and random seed, how to define components (specific protocol layers), how to assemble hosts and how to give them a basic input of actions, which they should perform during the simulation. Next, we will examine, how the simulations are run and what happens inside the simulator.

3.2 XML-Files with includes

The `config/includes`-subfolder and all its subfolders contain a lot of different predefined parts, that can be used to write a configuration file fast. Includes exploit the fact that most

party of a configuration file can be reused in other configuration files. So instead of writing them again and again, they are extracted in a separated file that can be included. The XML-File in Figure 9 has the basic properties as the one explained in the last section. As it is visible, the file is much shorter.

```

1 <Configuration xmlns:xi="http://www.w3.org/2001/XInclude">
2 <!-- In the "default"-section you may define variables to be used
   throughout your config by preceeding the name with a dollar sign -->
3 <Default>
4   <Variable name="seed" value="500" />
5   <Variable name="finishTime" value="120m" />
6   <Variable name="actions" value="config/chord-actions.dat" />
7   <Variable name="gnpDataFile" value="data/mod_measured_data.xml" />
8   <Variable name="churn" value="true" />
9 </Default>
10
11   <!-- SimulatorCore -->
12   <xi:include href="includes/simengine/Simulator.xml" />
13   <!-- NetLayer -->
14   <xi:include href="includes/network/ModularNetLayer.xml" />
15   <!-- TransLayer -->
16   <xi:include href="includes/transport/DefaultTransLayer.xml" />
17   <!-- Overlay -->
18   <xi:include href="includes/overlay/dht/chord/ChordNode.xml" />
19   <!-- Application -->
20   <xi:include href="includes/application/DHTLookupGenerator.xml" />
21   <!-- Monitor -->
22   <xi:include href="includes/analyzer/DefaultMonitor.xml" />
23   <!-- HostBuilder -->
24   <xi:include href="includes/hosts/DefaultHostBuilderWithChurn.xml" />
25   <!-- ChurnGenerator -->
26   <xi:include href="includes/churn/ExponentialChurn.xml" />
27   <!-- Scenario -->
28   <xi:include href="includes/scenario/DHTLookupGenerator.xml" />
29 </Configuration>

```

Listing 9: A XML-File with includes for chord.xml

A single xml file that is included contains mostly of a few lines of code. An example can be seen in Listing 10 for the DHT.xml. The parts are self-explanatory and likewise to the last section.

```

1 <?xml version='1.0' encoding='utf-8'?>
2 <Application
3   class="org.peerfact.impl.application.DHTLookupGenerator.
4   DHTLookupGeneratorFactory">
5   <Distribution
6     class="org.peerfact.impl.util.stats.distributions.
7     UniformDistribution"
8     min="30"
9     max="120" />
10 </Application>

```

As it can be seen, includes present an easy way to write a huge amount of different configuration files with only slight changes. numerous parts of the configuration files can be interchanged because of the already predefined includes. A good example for this is the `DHT.xml` in the `config` subfolder.

4 Simulation

In the previous sections we have seen how we can start and configure the simulator. Now we will look in detail at the simulation itself for a deeper understanding. PeerfactSim.KOM is an event based simulator. Everything what happens during the execution is ordered into a queue of events and handled one after the other. If new events were created during execution they would be arranged into the correct position of the queue (cf. Fig. 6). The management and execution of the simulation and the queue is handled by the `Simulator` and the `Scheduler`. New events can be added with the

```
scheduleEvent(Object content, long simulationTime, SimulationEventHandler
    handler, SimulationEvent.Type eventType)
```

method. The `simulationTime` describes the time from the beginning of the simulation in seconds and should be calculated with help of the time constants defined in the `Simulator`, the event `type` has the function to distinguish different events. The content contains arbitrary data and the handler is responsible for the event execution. Therefore it must implement the `SimulationEventHandler` interface with the `eventOccurred(SimulationEvent se)` method which is called when the event should be executed. The simulation time, type and data of the event is stored in the `SimulationEvent`. Often the `content` and the `handler` are represented by the same object.

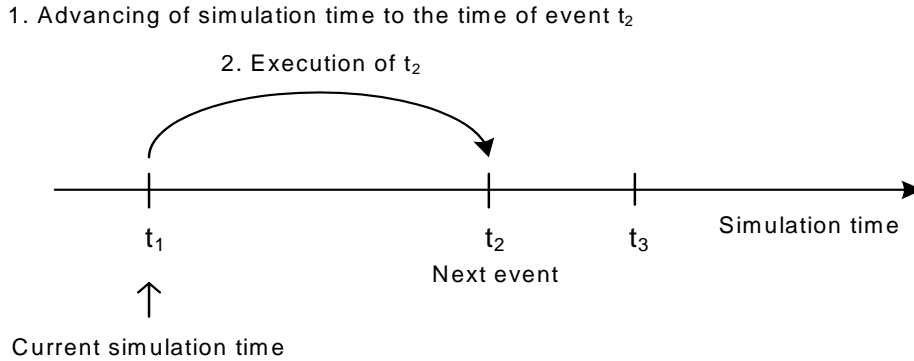


Figure 6: Example of the scheduling queue

There are several events which can be placed into the queue: On the one hand events required for the real simulation and on the other hand administrative events. In the first group operation executions and messages received events can be found as most important ones. We will handle them in detail in this section. The second group contains different task like starting and stopping of the simulator or of the monitor (cf. Sec. 6) and also events for

the underlying subnet and the churn model. By handling everything in the simulator as an event the correct execution can be guaranteed.

In this section we focus on the real simulation. First we present the overall concepts in PeerfactSim.KOM and afterwards we traverse the execution on two examples of chord.

The most important type for the real simulation are *operations*. Every action in an overlay network or an application is simulated in PeerfactSim.KOM as operation. In this object which is saved the executing host data is hold from the begin until the end of a process (cf. Fig. 7). In this chapter we will focus only on operations in the overlay layer, in an later section we present the concept of operations in the application layer (cf. Sec. 5). Examples for operations in the chord overlay are the join process as `JoinOperation` or the lookup process as `LookupOperation`.

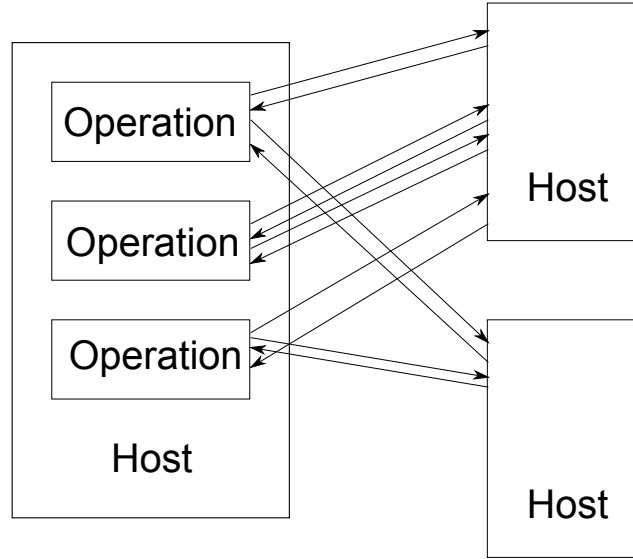


Figure 7: The operations concept

Technically all operations existing in the implementation inherit from the `AbstractOperation` which handles similarities like scheduling, handling of timeouts and it also acts as the event handler. The operation itself implements the `execute()`-method in which the operation specific action is handled.

There exists two different ways to generate the initial operations of the simulation: On the one hand by configuring different actions through an actions-file and on the other hand by different generators, for example by the `DHTLookupGenerator` which generates random lookups, for a DHT overlay. Both configurations were presented in the Sec. 3.

In the first way the actions file contains method calls which are executed at the destined times. For example in the case of the chord overlay, the `join()` method in the `ChordNode` will be called. This method creates the `JoinOperation` object and executes it at the respective time. In contrast to this, the generators call methods of the component on their own at random times depending on a model. In the case of the `DHTLookupGenerator` the `overlayNodeLookup()` method of the `ChordNode` will be called in different time slots. It creates the `LookupOperation` object and executes it.

As another next important part *messages* exist in the simulator. Every message implements the **Message** interface. The meaning of messages is to exchange information between the same layer on different hosts (cf. Fig. 8). The important message for our purpose is the **OverlayMessage** or in special the **ChordMessage**. It can be sent by using the **TransLayer**. While processing and sending the message it will be included in an **AbstractTransMessage** and **AbstractNetMessage**, but this should not be topic of this short guide. Similarly the message will arrive at the other host in the **TransLayer**. For processing each node has a **TransMessageListener** implemented, in the case of Chord this is the **ChordMessageHandler**, which processes the incoming messages in its **messageArrived()** method.

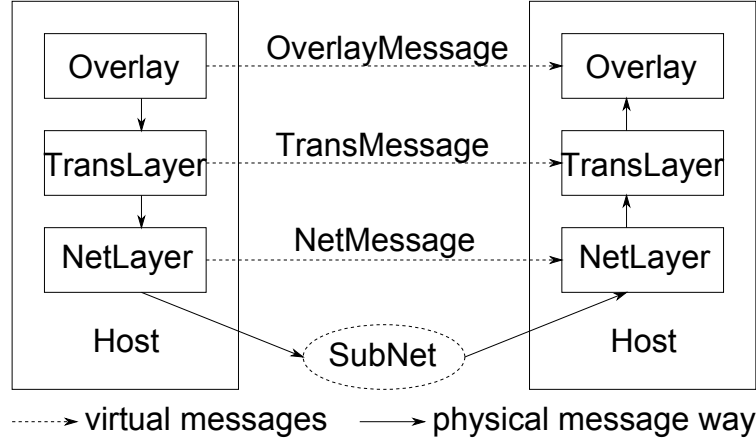


Figure 8: The messages concept

In addition to the operations and messages there exist so called *callbacks*. Since the simulation is event based, there is no way back to the caller of an action. Therefore the concept of callbacks exists on each level. Every operation can include an **OperationCallback**. After the operation has finished either the method **calledOperationSucceeded(Operation op)** or **calledOperationFailed(Operation op)** will be called in the caller.

Furthermore a **TransMessageCallback** can be added to a sent message. If a reply for this message arrives, it will be handled in the callback's **receive()** method instead in a **TransMessageListener**.

4.1 Examples for Chord

For a better comprehension on the presented concepts, we will present the execution of a **JoinOperation** and a **LookupOperation** in detail in a sequence diagram.

4.1.1 JoinOperation

The join operation in chord is processed every time a node wants to join the chord ring. The new one must know at least one node of the ring and sends a join message to it. The existing node starts a lookup operation to find the responsible node for the key of the new node. After a success it sends a join reply message to the new node with the found node.

As the last step the new node sends an handshake message to the found corresponding node and gets the finger table in the handshake reply to initiate its own one.

In the presented concept of PeerfactSim.KOM we have the following concrete implementation (cf. Fig. 9): The `JoinOperation` is initiated and started by the `join()` method of the new `ChordNode`. As first step the operation creates a `JoinMessage` and sends it to a random node in the chord ring with help of the `sendAndWait()` method in the `TransLayer`.

The message arrives at the receiver in the `messageArrived()` method of the `ChordMessageHandler`. Here a `LookupOperation` for the corresponding key will be created and proceeded through the `overlayNodeLookup()` method of the `ChordNode`. After the successful operation the `HandshakeCallback` creates a `JoinReply` with the information about the found node and uses the `sendReply()` of the `TransLayer` for the answer.

In the initiating node the `JoinOperation` was defined as callback for the join message and therefore the `JoinReply` arrives directly in the `JoinOperation`. The operation prepares a `HandshakeMessage` and sends it to the found responsible node where it is handled in the `messageArrived()` method of the `ChordMessageHandler`. A `HandshakeReply` with the finger table is created and send back to the joining node which can finish the `JoinOperation` successfully. As last step the `joinOperationFinished()` method of the `ChordNode` is called in which the routing table for the new node is created with the information of the `HandshakeReply`.

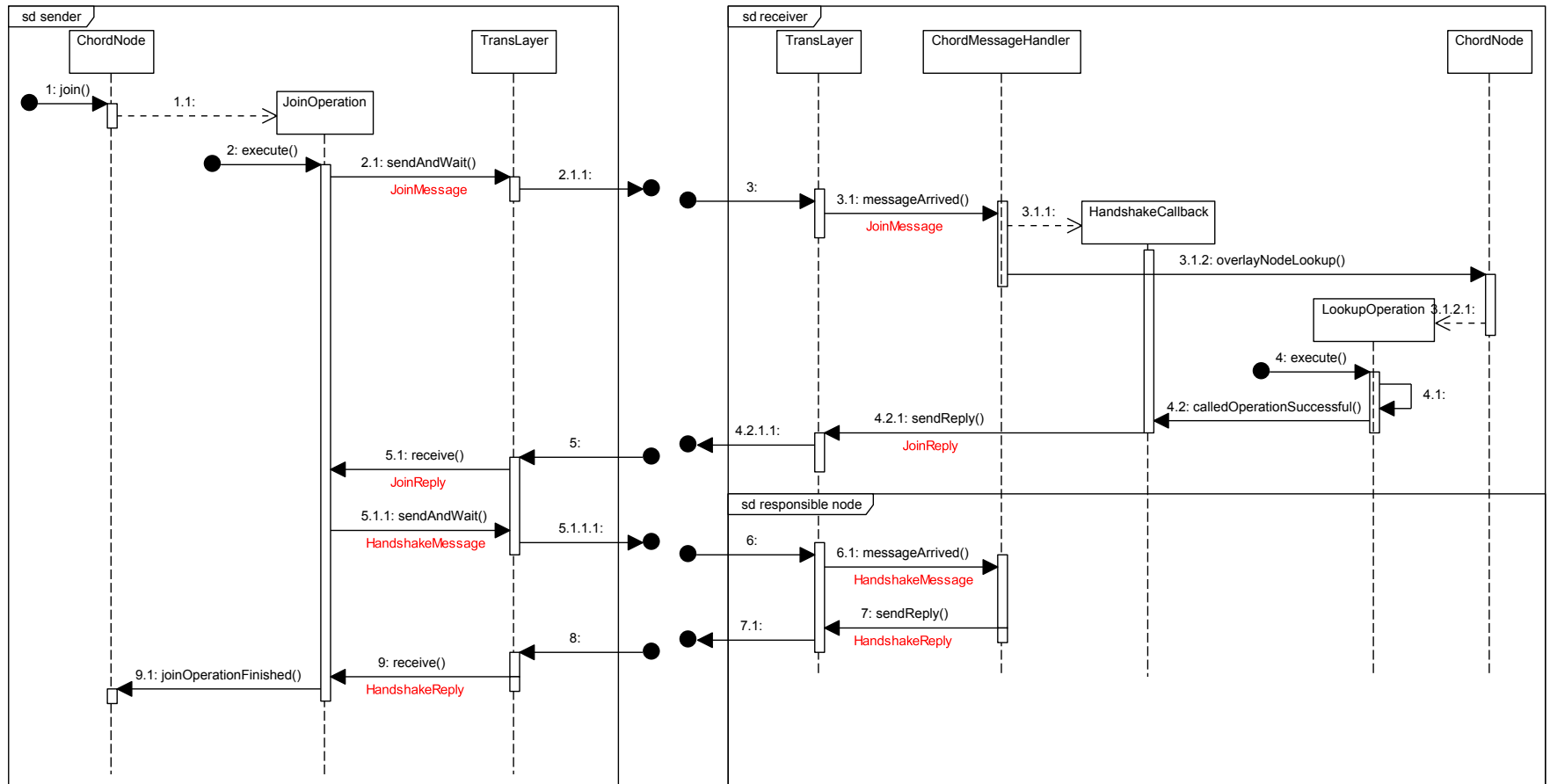


Figure 9: Sequence diagramme of JoinOperation

4.1.2 LookupOperation

As a second important operation within chord, the **LookupOperation** is presented. The node who wants to look up a certain key sends a message to another node corresponding on its finger table⁴. If the receiver is responsible itself it sends a reply, otherwise it sends the lookup message to the next node regulated by its finger table. All messages are approved by an acknowledgement.

In the implementation (cf. Fig. 10) the lookup starts with the **overlayNodeLookup()** method of a **ChordNode**. Here the **LookupOperation** is created. In the execution the operation creates a **MessageTimer** object as callback to receive the acknowledgement. Then it sends the **LookupMessage** through the **TransLayer**, the receiver is found by the routing table.

At the receiver the message arrives in the **messageArrived()** method of the **ChordMessageHandler**. If the node is not responsible it sends a new **LookupMessage** to the next node depending on the routing table. In the example the node is responsible. It creates also a **MessageTimer** as callback and sends a **LookupReply** through the **sendAndWait()** method of the **TransLayer**. Independent from the action the node sends also an **AckMessage** as reply for the incoming message.

The original sender receives the answer in the **messageArrived()** method of the **ChordMessageHandler** and can finish the **LookupOperation** by using the **deliverResult()** method. It also sends an **AckMessage** as reply.

The **AckMessages** arrive in the **receive()** methods of the created **MessageTimers** to make sure that each message was correctly delivered, otherwise a time out occurs and another transmission will be started.

⁴Chord works with a finger table as routing table in each node. This table contains references to the next nodes in the routing in larger intervals to manage bigger steps than the normal iteration through the ring.

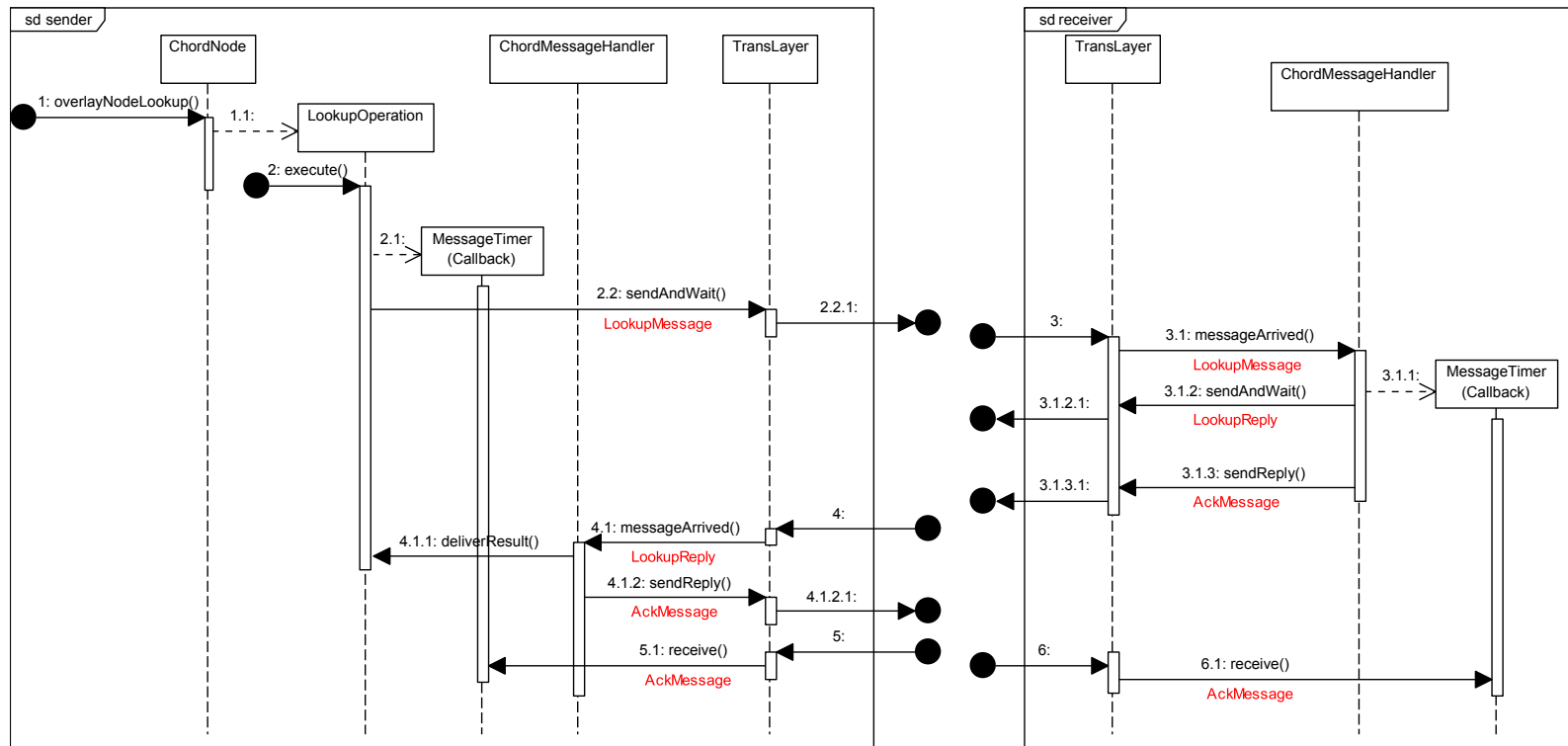


Figure 10: Sequence diagramme of LookupOperation

5 Overlays and Applications

In the previous section the concept of the simulator was presented with focus on the overlay layer. All explained operations were initiated on this layer. Besides the simulation of the actions in the overlay layer PeerfactSim.KOM offers the possibility to simulate different applications on top of the peer-to-peer overlay layer. At the moment a default **Filesharing** application and different **LookupGenerators** are implemented.

The operation concept visualized in the previous section is also present in this layer. With help of generators or actions defined in an actions file filesharing specific operations can be scheduled. Many of the applications operations use functions of the underlying overlay, for example the presented `join()` or `overlayNodeLookup()` method of the **ChordNode**. As a consequence we get nested operations on the different layers. The reader of this guide should be able to understand the implementation of the application with the knowledge of the previous section.

6 Analyzer

The main reason for using a simulator is the generation of statistics on the reviewed aspects. Therefore PeerfactSim.KOM has the possibility to collect data and calculate results.

In analogy to the layered structure of the simulator there exist different analyzer types (cf. Fig. 11). For the overall monitoring of the simulation an **Analyzer** can be defined which calculates statistics independent of the current simulation events (for example it counts the available hosts at different times). The operation handling, especially the start and end of the separate operations, can be observed by different **OperationAnalyzer**. The overlay, transport and network layer can be monitored by a **DHTOverlayAnalyzer** or a **KBROverlayAnalyzer** or a **UnstructuredOverlayAnalyzer** (message forwarding and query monitoring), **TransAnalyzer** (sending and receiving of messages), **NetAnalyzer** (sending, receiving and dropping of messages) or **ConnectivityAnalyzer** (online and offline status of hosts). As last type the **ChurnAnalyzer** monitors the modelled churning.

To use analyzers in the simulation, the **Monitor**-part with analyzers must be defined in the configuration (cf. Sec. 3). The **DefaultMonitor** is the basis of the whole monitoring and analysis process, every configured analyzer is registered to it. The monitoring will be started and stopped by different events in the simulation, the time is configured by the **start** and **end** attributes in the configuration file.

The different types of analyzers are represented as interfaces in the implementation (cf. Fig. 12). All analyzers inherit from the **Analyzer** which only provides the **start()** and **stop()** methods. These two are called by the monitor at the beginning and end of the monitoring. The analyzer can do their own initialization (for example preparing of files as in the **AbstractFileAnalyzer**). Most of the analyzers create themselves events in the simulation queue to do different jobs periodically (for example write statistics to a file). Some analyzers only use these periodic tasks to generate data with help of an oracle⁵ which has general knowledge about all hosts.

Furthermore specific interfaces for each analyzer type exist and can be used. In addition

⁵The **GlobalOracle** can be configured in the configuration file and has global knowledge about all hosts for analyzing.

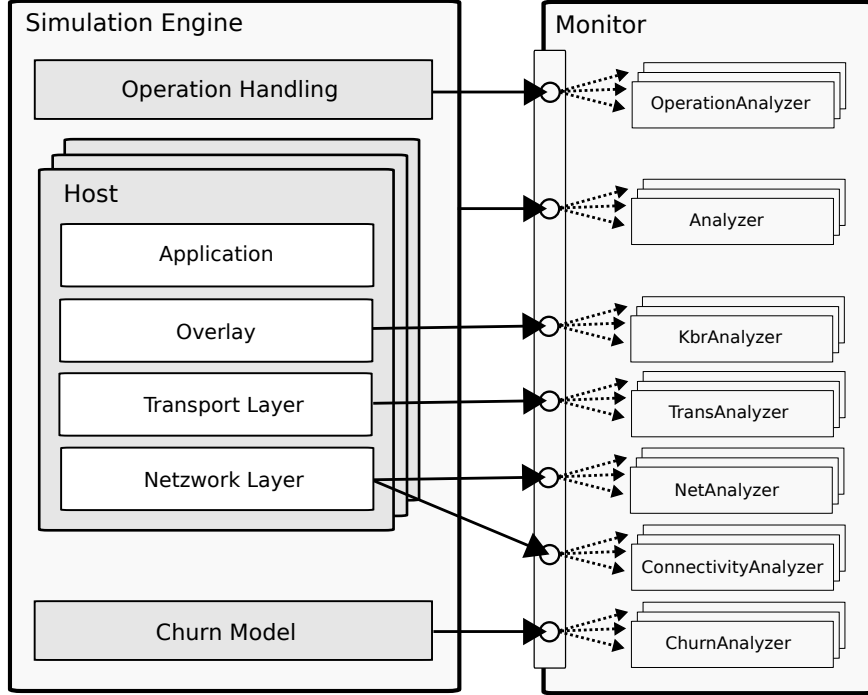


Figure 11: Overview about the analyzers

to the `start()` and `stop()` methods they declare different event specific handler methods. The `DefaultMonitor` passes the monitored events to the corresponding registered analyzers which use the data to generate statistics.

All analyzer implementations which inherit from the `AbstractFileMetricAnalyzer` use a metric concept for collecting data. The analyzer itself only add data to the different used metrics, for example a `CounterMetric` or a `StatisticMetric`. The base analyzer is responsible for the analysis and output of the statistics.

A special analyzer exists in the `VisAnalyzer`. It implements four interfaces (`ConnectivityAnalyzer`, `NetAnalyzer`, `TransAnalyzer` and `OperationAnalyzer`) and collects the data like all other analyzers. The difference is usage of the data. Instead of generate statistics in files all data is hold in memory and presented in the visualization window after the simulation has finished. From here it can be saved as visualization file.

6.1 Example for chord

After an overview on all existing analyzers we will show the monitoring on the example of the `JoinOperation` presented in Sec. 4.1. The steps in the explanation relegate to the sequence diagrams (cf. Fig. 9).

The first monitor call is executed when the `LookupOperation` starts (step 2). This event is the start of every operation and calls the `operationInitiated()` method of the `DefaultMonitor` and of each registered `OperationAnalyzer`. While sending the `JoinMessage` (step 2.1) the methods `transMsgSent()` and `netMsgSend()` are called in the responsible analysers.

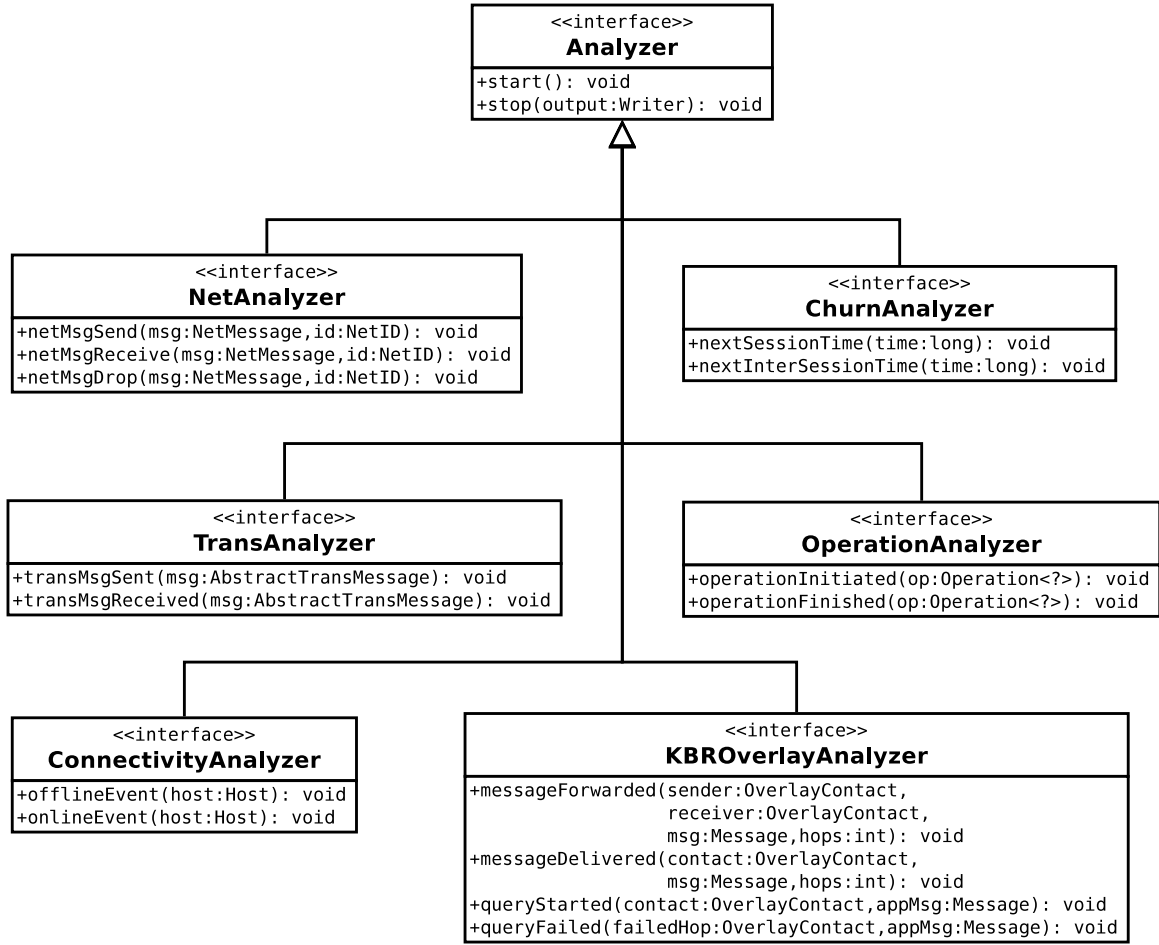


Figure 12: The interfaces for different analyzers

At the receiver node the `netMsgReceive()` and `transMsgReceived()` methods are called in step 3. While handling the messages the `ChordMessageHandler` provides its status by calling `messageForwarded()` and `messageDelivered()` methods of the `DefaultDHTOverlayAnalyzer` (step 3.1). During the further execution the same analyzer methods are called at the corresponding steps. After the successful join the `operationFinished()` method is called at the end of every operation (step 9.1).

Furthermore `offlineEvent()` and `onlineEvent()` are called depending on the churn model.

7 Further Information

This was only a short overview of the functionality of PeerfactSim.KOM. There are several possibilities to delve deeper into the matter.

First of all the website <http://www.peerfact.org> presents all up-to-date information concerning the simulator. Next to the current release you can find several documents, video tutorials, a forum and other materials there. This is also the first way to get in contact with

the developers.

For a deep look in the underlying technique, especially at the lower layers, you should read the full documentation, also available at the homepage.

Furthermore you can go through the source code to get a deep insight with help of many comments you find in it.