PeerfactSim.KOM: A Large-Scale Simulation Framework for Peer-to-Peer Systems

Dominik Stingl, Christian Gross, Julius Rückert, Leonhard Nobach, Sebastian Kaune, Konstantin Pussep

TECHNISCHE UNIVERSITÄT DARMSTADT

KOM – Multimedia Communications Lab

PeerfactSim.KOM:
A Large-Scale Simulation Framework for Peer-to-Peer Systems
Dominik Stingl, Christian Gross, Julius Rückert, Leonhard Nobach, Sebastian Kaune, Konstantin Pussep
http://www.kom.tu-darmstadt.de

Technische Universität Darmstadt
Department of Electrical Engineering and Information Technology
Department of Computer Science (Adjunct Professor)

Multimedia Communications Lab (KOM)
Prof. Dr.-Ing. Ralf Steinmetz

Contents

List of Figures

List of Tables

Listings

1 Introduction

This documentation describes the java-based simulator PeerfactSim.KOM, which is a project, launched at Multimedia Communications Lab (KOM) for the simulation of large-scale peer-to-peer system. The idea of this project is to create a simulator, that fits to a large amount of use cases and enables the simulation of different scenarios in the area of peer-to-peer (P2P). In order to utilize PeerfactSim.KOM for simulations as well as to allow for extending the simulator, we provide this documentation, that on the one hand details the architecture and implemented concepts, while on the other hand, provides useful information for further development and improvement. As a result, PeerfactSim.KOM that was started a couple of years ago, is always in progress and kept up-to-date. Due to this fact as well as to the amount of former and present developers, we consider this document as a kind of open document, where every developer contributes a documentation for the component, that he implemented. So this documentation is just a collection of different, small documents highlighting only one or a few components of the complete simulator.

In the subsequent sections of this chapter we highlight the essential design decisions and reasons for the chosen architecture of the simulator. Besides, we give a brief overview of the components and their functionality, of which the simulator consists as well as sketch the applied models, e.g. for communication, event processing and so on. The rest of this documentation, that also refines the overview of this chapter is structured as followed. Chapter 2 gives again an overview of PeerfactSim.KOM and details some general concepts regarding the layered architecture, while Chapter 3 deals with the single layers and their functionality. Having described the parts of which PeerfactSim.KOM consists, the remaining two chapters sketch the interaction with the simulator. Therefore, Chapter 5 shows how simulations can be configured and executed and Chapter 6 introduces an option to use a visualization component.

1.1 General considerations and design decisions

When designing the architecture, our fundamental idea has been at the one side to come up with a concept that is easily understandable, clearly structured and partly related to the ISO/OSI basic reference model [3], but on the other side capable to support architectures which do not necessarily follow a layered approach. Since we cannot foresee the requirements of future P2P system research, our aim is to provide a general-purpose P2P simulator, which is not dedicated to any specific P2P architecture or system and should not be biased by a predominantly linear protocol stack.

For this reason a framework-like approach has been designed that is based on the concept of plug-ins to which we also refer as *components* or *elements* of the simulator in the following. As a first step, many different architectures of P2P systems like overlay networks (structured, unstructured, flat, hierarchical, hybrid) or content distribution systems have been analyzed to capture their requirements. Following this step, the most important entities have been identified and developed as well-defined components in the software architecture of the simulator. The idea behind the components or the aforementioned plug-ins is that they describe individual functionalities, which each deal with unique aspects related to the context they are embedded in and provide services that can be used by other components. To concretize these design decisions of the different elements, of their integration and interaction, a well-defined interface is supplied for each identified component to offer the required functionality. Additionally, for each interface either a default implementation (in cases where it offers general purpose functionality suitable for any P2P approach) or an abstract base implementation, a so called *skeletal implementation* with general functionality is provided. This skeletal implementation already includes necessary code to make life for the developer as easy as possible. Based on these interfaces and abstract classes, developers can customize the concrete implementations of each interface either by replacing or extending a default component

implementation or by providing specific extensions to a skeletal implementation. The process is straightforward since there is a well-defined interface that can be seamlessly referenced in the implementation of other components. Hence, the interaction among the default or the customized components takes place through their interfaces, making the interaction process agnostic of implementation details. Figure 1.1 illustrates this *Component design pattern*. The following chapters, that deal with the layered architecture (e.g. Chapter 3) or directly address specific components (e.g. Chapter 4) will pick up this Component design pattern and provide the respective names of the defining interfaces, default implementations and extendable, abstract classes where required. Having these concepts and contemplated patterns in mind, we display the basic package structure, the simulator consists of, and which reflects again the classification in the defining interfaces of an identified component and in the abstract, default or concrete classes:

- `de.tud.kom.p2psim.api`
- `de.tud.kom.p2psim.impl`

The former contains the API which is further divided into sub packages for each functional layer and component. The same structure is reflected in the latter in which one can find the default or abstract implementations of important components that will be described in the following sections.

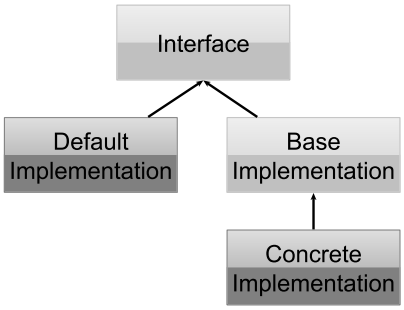


Figure 1.1: Component Design Pattern

1.2 Architectural Overview

After sketching the essential design decisions and reasons for the chosen architecture of the simulator, this section provides a brief overview of the simulator's architecture, which tries to address effectively the complexity of P2P systems. Therefore, an analysis of the functionality of the different systems has been performed to divide them into distinguishable layers. This separation forms the basic architecture of PeerfactSim.KOM, that can be currently divided into two parts: the functionality layers and simulation engine. Figure 1.2 depicts the basic architecture of the PeerfactSim.KOM.

To sum up, each layer provides well-defined interfaces to express its individual functionalities, operations and provided services. More specifically, core components for each layer have been identified and offer different services that can be used by other layers or components. When deciding how many layers to include in our simulation framework and what each one should do, one of the most important considerations has been defining clear interfaces between the layers. Doing so, in turn, requires that each identified layer perform a specific collection of well-understood functions. In addition to minimizing the amount of information, that must be passed between the different layers, clear cut interfaces also make it simpler to replace the implementation (=internally used components) of one layer with a completely different one. All that is required of the new implementation is that it must offer exactly the same set of services as the old implementation did. This can be accomplished, if the new implementation extends an abstract class or directly implements a well-defined interface of a certain component (We refer to the Component design pattern in Section 1.1).

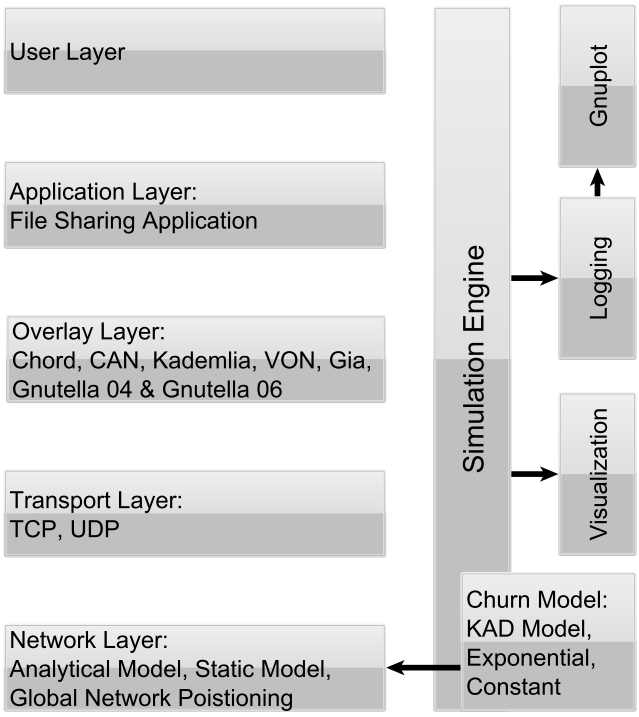


Figure 1.2: Layered Architecture

Based on the layered architecture depicted in Figure 1.2, we focus on the functional layers of the simulator and sketch their purposes in the following. The network layer maintains a number of services which embody the network level functionality of the bottom three layers of the ISO/OSI model [3]. There, it is possible to easily select different network service models to vary the degree of abstraction dependent on the purpose of a simulation. Section 3.1 explains its functionality in detail, while Section 3.2 deals with the transport layer, which sits above the network layer and encapsulates the details of the data transfer between endpoints within the simulation framework. The overlay layer, highlighted in Section 3.3, contains a number of overlay communication protocols and comprises algorithms such as overlay message routing. In the scope of simulating P2P systems, this layer is one of the most important layers of the simulator, since it comprises the aforementioned protocols, that realize and implement the different overlay networks (structured, unstructured, flat, hierarchical, hybrid) and build the core of P2P systems. Above the overlay layer, the application layer aims to provide a general interface to implement and simulate the functionality of common P2P applications within a complete P2P system (See Section 3.4). Finally, the user layer, addressed in Section **??**, captures the action taken by the users to model different behaviors. Altogether, these functional layers are incorporated in a so called *host* which acts as an endpoint in a P2P system. To clarify the usage of the term *host* for the rest of the documentation, it will be treated as a synonym for *peer* and both terms are exchangeable in respect of their meaning.
Besides these functional layers, there also exists the discrete-event based simulation engine, which represents the basis layer of the simulator and is examined in Section 2.1.

As far as the interaction between the layers is considered, it takes place, either using function calls or with the use of events in the system. For this purpose, some functional layers as well as the simulation engine are offering interfaces, which other components can use to register their event handlers and thereby can interact with each other. In case of information exchange or data transfer between different hosts, the simulation framework doesn't allow for a direct transfer of data from layer x on one host to another layer x on another host. Instead, like in reality, each layer passes data and control information to the layer below, until the simulation engine is reached. In Figure 1.3, virtual communication is shown by the dotted lines and simulated, physical communication by solid lines. To substantiate the information

exchange within the simulator and to concretize the displayed content of Figure 1.3, we state that PeerfactSim.KOM is a message level simulator, where communication among the layers is provided by using messages. So every time two hosts want to communicate with each other, a message is produced by an application or overlay layer and given to the transport layer for transmission. The transport layer puts the *TransportAddress* of the destination, which comprises its *Port* and *NetID*, as additional information to the message. This information is necessary to allow that the correct remote application or overlay on the destination host gets the message. The network layer concatenates the resulting message unit with the NetID of the sending host and passes this composed message to the simulation engine. At the receiving host, the message moves upward from layer to layer using the event system described above with the additional information being stripped off as it progresses. The detailed processing of the messages will be explained in Subsection 2.2.2 which focuses on the communication between the different layers within a Host when a message is sent or received. In this context, we also hightlight the utilisation of the aforementioned interfaces and event handlers. A further description of message processing will be given in Section 3.1 which focuses on the transmission of a message between two host over the network.

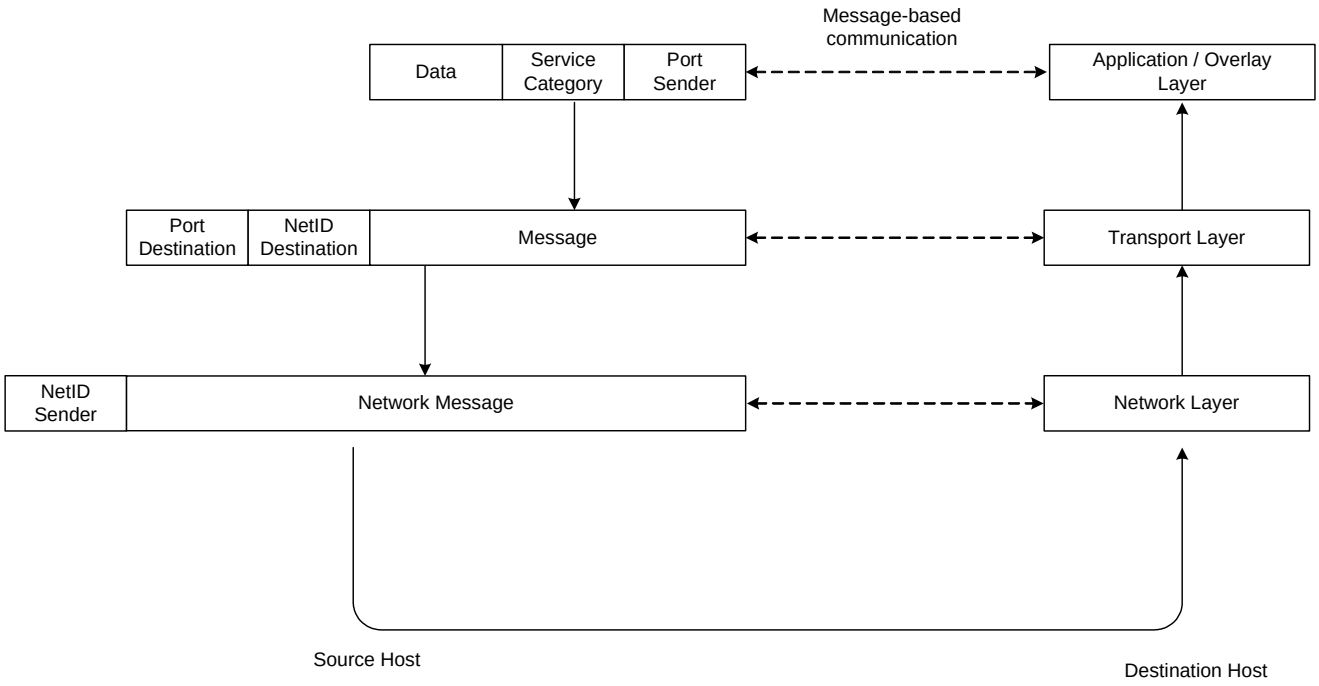


Figure 1.3: Example Information flow supporting virtual communication in Application/Overlay layer

2 Overview of the simulator

As already sketched in the previous Section 1.2, this chapter deals with the structure of the simulator as well, but provides more details on the simulation engine (Section 2.1) and the layered architecture as well as general concepts for the interaction between the simulation engine and the layered architecture (Section 2.2). In fact, the latter section also examines the communication between the single layers, how the status of a peer/Host changes and can be recognized and what the concept of Operations within PeerfactSim.KOM is meant for and how it can be used for a simulation.

2.1 Simulation Engine

In general, one can say all discrete-event based simulations have a common structure. Regardless of the peer-to-peer system being modeled, the simulation will contain some of the basic components which will be briefly explained below.

Event Scheduler This component is one of the most frequently executed during a simulation. The `scheduleEvent(...)`-method is executed before every event can occur and it may be called several times within one event to trigger other new events. The operations on it are as follows:

- Schedule event x at time t
- Cancel a previously scheduled event x
- Get current simulation time (see description below)
- Set finish time t in order to finish the simulation on a specific point in time

Simulation Time and a time-advancing mechanism Each simulation has a global variable representing the simulation time. The scheduler is responsible for advancing this time which will be incremented automatically to the time of the next earliest occurring event

Event Queue Event scheduling is done by keeping an ordered list of future SimulationEvents waiting to happen. Each simulation event contains the time at which it should occur and a reference to a `SimulationEventHandler` that will be informed about the occurrence at that time.

In order to comprehend the details of discrete event based simulations, it is necessary to understand the scheduler's way of operation. In each atomic simulation step, the scheduler fetches the next earliest occurring event from the queue, calls its correspondent event handler performing various actions. Afterwards, the next events are processed one after another until the event queue is empty or the simulation end time is reach. For example, consider a scenario where we have a join request at simulation time $t_1 = 1$ and a second one at $t_2 = 1000$. Then, the first event is executed, and the simulation time is set to 1. Immediately thereafter, the second request is being executed, and the simulation time increased to 1000. Notice that the term "discrete" does not apply to the time values. This means the simulation engine doesn't wait for 999 time units in real time.

Within the simulator, the simulation engine is responsible for managing all components that are involved within a simulation. So the engine, in fact the event Queue, contains events, which are disseminated to the corresponding component (e.g. a respective layer, the churn generator etc.) in order trigger a respective action. Therefore, each event contains information about the component, the event is addressed to, as well as information about the action, the event will trigger. In order to be able to schedule and execute an event as a component, the component must implement the interface `SimulationEventHandler`[1]. By implementing this interface, a component sets itself as

[1] `de.tud.kom.p2psim.api.simengine`

the owner and receiver of the event. If the component calls the method stated in Listing 2.1 of the class `Simulator`[2], the component registers itself as `SimulationEventHandler` and becomes the owner and receiver of the event. By implementing `SimulationEventHandler`, a component must implement `eventOccurred(SimulationEvent se)`, which receives the event from the scheduler in order to execute it. Based on the information of the event, the method executes the respective actions.

After having described the general concept of an event-based scheduling engine and its processing of events as well as interaction with a component of a simulation, the following section details the other main part of the simulator, represented by the layered architecture.

```
1 public static void scheduleEvent(Object content, long simulationTime,
2       SimulationEventHandler handler, SimulationEvent.Type eventType)
```

Listing 2.1: Method for scheduling an event. numbers

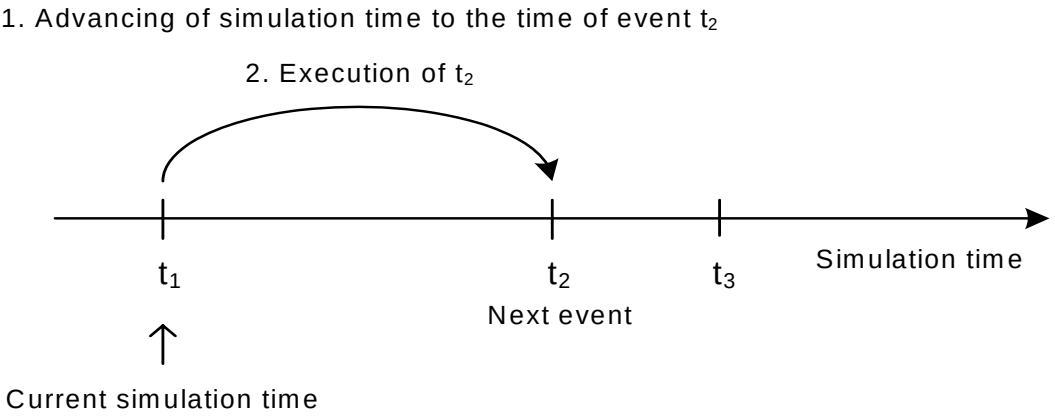


Figure 2.1: Advancing of simulation time in discrete event-based simulations

2.2 General concepts of and for the layered architecture

The layered architecture describes the components, the simulator currently provides, and which may be simulated during a scenario. While Chapter 3 focuses more on the details regarding the offered functionality as well as the implementation for a specific layer (e.g. Chord, Kademlia etc. for the Overlay Layer) and finally justifies why these layers have been chosen to be simulated, this section describes some general concepts which mainly apply to all layers and must be taken into consideration when using a specific implementation for a layer for a simulation.

2.2.1 Meaning of the Host component

It is important to understand that during a simulation every peer is represented by a *Host* as depicted in Figure 2.2. In the current version, this Host is implemented by the class `DefaultHost`[3] and comprises all layers of the layered architecture that were specified by the configuration file for a simulation (We refer to Section 5.1 for more information about the configuration). Besides the mentioned layers, the Host also comprises a further component called *HostProperties* which contains general information about the Host like up- and download bandwidth, currently consumed bandwidth, a possible `groupID`. Depending on the requirements of the simulation, the HostProperties can be extended with further properties like CPU power, storage, RAM etc. During the configuration, every layer which is added to a host is registered

[2] `de.tud.kom.p2psim.impl.simengine`
[3] `de.tud.kom.p2psim.impl.common`

by the Host and in return registers the Host as reference. By this mutual registration the Host is aware of all its layers and components and can provide general information to them, while every layer and component has access to the host.

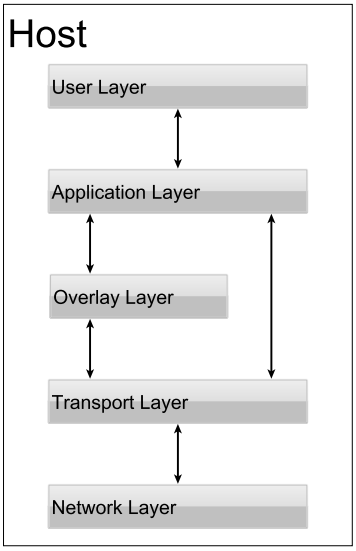


Figure 2.2: Representation of a peer during a simulation as Host

During a simulation, the Event Scheduler distributes the fetched events of a queue to the different layers of the simulated Hosts which react accordingly. Depending on the type of the event and its impact for the remaining layers, the Host can be used to provide the information or the new state beyond the layers. Actually, the Host utilises this mechanism to provide information about the current network state. If the Network Layer of a Host becomes connected or disconnected from the network, this state information is provided to the remaining layers that are interested in the current network state. Based on the information, a layer can trigger respective actions (e.g. if a Host comes online, the Overlay Layer tries to reconnect to the overlay, if a Host gets disconnected, the Overlay Layer should stop executing the overlay protocol). In order to get such kind of information, the interested layer must separately register itself at the Host. Listing 2.2 gives an example of a component that registers the Host and additionally adds itself to the list of components which shall receive the information about the status of the network connectivity.

```
1 public void setHost(Host host) {
2   this.host = host;
3   this.host.getProperties().addConnectivityListener(this);
4 }
```

Listing 2.2: A component registers the Host and adds itself to list in order to be informed about changes regarding the status of the network connectivity

To add a component to the aforementioned list, the component must implement the method `public void connectivityChanged(ConnectivityEvent ce)` from the interface `ConnectivityListener`[4], which handles the provided information and reacts accordingly.

Based on the example of the connectivity state of the Network Layer, every information that is important for the Host or other layers can be disseminated by such an integrated architecture which is based on the Observer Pattern [10].

[4] `de.tud.kom.p2psim.api.common`

2.2.2 Sending and receiving messages inside the Host

As already mentioned during Section 1.2 and displayed by Figure 1.3, the simulator uses messages instead of packets. Within the sending Host, the message is pushed from one layer to the next one below, using the appropriate send-method of the respective layer. The Host, receiving the messages uses an architecture based on the Observer Pattern [10] to push the information to the layers one or several layers above. So, every layer which is interested in getting messages from a layer below, must implement the `messageArrived`-method from the respective Listener (`NetMessageListener`[5] or `TransMessageListener`[6]) and register itself at that layer, calling the `addXXXListener`-method. The existing interfaces and methods for registering are summarized in the following:

- **Network Layer** Every layer (actually just the Transport Layer) can register itself at the Network Layer by calling `public void addNetMsgListener(NetMessageListener listener)` of the class that implements the `NetLayer`-interface[7]. A higher layer which registers itself at that layer must implement the `NetMessageListener`-interface in order to provide itself as argument for the aforementioned method.
- **Transport Layer** Every layer can register itself at the Transport Layer by calling `addTransMsgListener(TransMessageListener receiver, short port)` of the class that implements the `TransLayer`-interface[8]. A higher layer which registers itself at that layer must implement the `TransMessageListener`-interface. The `addTransMsgListener`-method additionally requires a port as second argument to allow for distributing the incoming messages based on the port.

2.2.3 Idea, usage and implementation of an Operation within PeerfactSim.KOM

Within this subsection, we focus on the concept of an *Operation* in the simulator and describe the idea and implementation of it. In general, Operations can be used for for common actions in a distributed system. Typically, if some action should take place in a component of a host, the action will be represented by an operation object providing the required functionality. Their main scope is to perform actions, sequences of actions or cyclic routines which in turn consist of actions or sequences of actions. As an example for every kind of Operation, we state the following examples:

Join Operation A peer is interested in joining the overlay and tries to join it by invoking the respective Operation. Depending on the outcome, if any, the peer successfully joins the overlay or fails and tries it again for a certain amount of time.

Store Operation A peer is wants to download a file, therefore it searches for the responsible peer and downloads the file from it.

Stabilize Operation This Operation must be periodically executed by a peer in order to e.g. refresh its routing table. The periodic stabilization may just consist of a single operation like a Lookup-Operation or contain a sequence of different operations.

As already indicated by the first example, an Operation has three different outcomes:

No result In effect, there is always a result returned, but in this case it does not matter.

Successful execution The Operation was executed in time and returned a result. The returned result may either be useful for the receiver or not.

Unsuccessful execution The Operation timed out and could not be finished.

[5] `de.tud.kom.p2psim.api.network`
[6] `de.tud.kom.p2psim.api.transport`
[7] `de.tud.kom.p2psim.api.network`
[8] `de.tud.kom.p2psim.api.transport`

While the difference between the first and the two remaining outcomes mainly depends on the utilization and interpretation of the user, the outcome of an Operation (successful or unsuccessful) is returned by the Operation itself. To help implementing Operations within PeerfactSim.KOM, the interface `Operation`[9] which contains most of the required methods for starting, observing and finishing an Operation is already implement by the abstract class `AbstractOperation`[10]. To determine the state and the outcome of an Operation, `AbstractOperation` carries two Boolean fields `error` and `finish` to express its current state indicating, if the operation has already been finished with or without success.

To start an Operation, it must be scheduled using one of the three methods provided by `AbstractOperation`:

`scheduleImmediately()` Schedules the operation immediately without any delay
`scheduleWithDelay(long delay)` Schedules the operation with an delay relative to the current simulation time
`scheduleAtTime(long time)` Schedules the operation to an absolutely point in time

When the event, containing the Operation, is fetched by the Event Scheduler and processed, the abstract method `protected abstract void execute()` from `AbstractOperation` is called. Within a concrete implementation of `AbstractOperation`, `execute()` contains the whole logic and instructions of the Operation which are then executed. Inside this method, one can specify the timeout for the Operation by invoking which schedules the respective expiration date (See Listing 2.3). If this event is fired by the Event Scheduler, the Operation is automatically finished with an unsuccessful outcome. Another possibility to finish an Operation, can be realized by invoking the `protected void operationFinished(boolean success)`-method of `AbstractOperation`. Depending on the result of the finished Operation, one can define the outcome by setting the `success`-argument of the method accordingly.

```
1 protected void execute() {
2   // Schedule the timeout for the operation
3   scheduleOperationTimeout(timeout);
4   // The logic and instructions of the concrete Operation
5   overlayNode.doSomeOperation();
6 }
```

Listing 2.3: The implementation of an `execute`-method within a concrete Operation extending of `AbstractOperation`

In order to react on the events, fired by the Event Scheduler, `AbstractOperation` implements the method `public void eventOccurred(SimulationEvent se)` of the `SimulationEventHandler`-interface as displayed in Listing 2.4. Depending on the type of the event, the Operation is started by invoking the `execute()`-method or a timeout occurred that changes the state of the Operation to an unsuccessful outcome and aborts it.

```
1 public void eventOccurred(SimulationEvent se) {
2   if (!isFinished() && se.getType() == SimulationEvent.Type.TIMEOUT_EXPIRED) {
3     operationTimeoutOccured();
4   } else if (se.getType() == SimulationEvent.Type.OPERATION_EXECUTE &&
      se.getData() == this) {
5     Simulator.getMonitor().operationInitiated(this);
6     execute();
7   }
8 }
```

Listing 2.4: Implementation of `public void eventOccurred(SimulationEvent se)` within `AbstractOperation`

[9] `de.tud.kom.p2psim.api.common`
[10] `de.tud.kom.p2psim.impl.common`

As pointed out in the previous paragraph, an Operation always has an outcome which either can be utilised for further processing (e.g. the result of a Lookup Operation) or which can be neglected. Regarding the first case, the question arises, how a result of an Operation can be used by a component for further processing within a discrete event-based simulator. Since the caller of an Operation schedules the execution of the operation for a certain point in time and therefore passes the Operation to the Event Queue while the rest its program is directly processed, the result of the Operation is not available for the subsequent processing. Listing 2.5 therefore shows an example of a typical Java-program, where a normal operation is executed while its result can be directly used for further processing.

```
1  public void useLookupResult(OverlayKey key) {
2    // An operation is executed for retrieving the responsible peer for a key
3    OverlayID id = operation.lookupOverlayKey(key);
4    // Depending on the result, the rest of this method is executed
5    if (id == null) {
6      restartLookup(key);
7    } else {
8      useID(id);
9    }
10 }
```

Listing 2.5: Example of an operation call in a typical Java-progam

To solve this problem, PeerfactSim.KOM introduces the concept of asynchronous *Operation Calls* that allows for invoking an Operation at a certain point in time, while the result can be used later on, when the Operation has finished. Therefore, each Operation must be endued with a class that implements the `OperationCallback`-interface[11]. This interface provides the two methods listed in Listing 2.6 that are invoked by `AbstractOperation` when an Operation was automatically or manually finished.

```
1 // Called if the operation failed.
2 public void calledOperationFailed(Operation<T> op);
3
4 // Called if the operation was finished successfully.
5 public void calledOperationSucceeded(Operation<T> op);
```

Listing 2.6: The offered methods of the `OperationCallback`-interface

Referring to the example given by Listing 2.5, one could implement this interface to achieve the same functionality as shown in Listing 2.7. In fact, this listings states a valid example for a Lookup-Operation in PeerfactSim.KOM. The newly instantiated class `LookupOperation` is counterpart to the call within Listing 2.5

```
1  public void useLookupResult(final OverlayKey key) {
2    // An operation is executed for retrieving the responsible peer for a key
3    LookupOperation op = new LookupOperation(key, new OperationCallback<Object>() {
4
5      public void calledOperationFailed(Operation<Object> op) {
6        restartLookup(key);
7      }
8
9      public void calledOperationSucceeded(Operation<Object> op) {
10       useID(op.getResult);
11     }
12   });
13 }
```

Listing 2.7: Example of an asynchronous operation call in PeerfactSim.KOM

[11] `de.tud.kom.p2psim.api.common`

In case that a result of an Operation is not needed and therefore can be neglected, PeerfactSim.KOM provides the class `Operations`[12] which allows to utilise and schedule an empty `OperationCallback`.

[12] `de.tud.kom.p2psim.impl.common`

3 Functional Layers

This chapter gives a basic documentation about the functional layers of the simulator. We first describe the layered architecture and its purpose, then we go into detail about the single layers and their implementations. We give an overview about the existing implementations for a layer, explain their features and how they can be used for a simulation. The idea behind the layered architecture is to allow for an independent configuration and for the possibility of exchanging single layers that simulated Hosts consist of. Since this simulator focuses on the simulation of P2P networks, it only contains layers that are considered to be useful for such simulations. This means for example that the "lowest" layer modeled, according to the ISO/OSI basic reference model [3], is layer three (the Network Layer). Layer one and two (Physical and Data Link Layer) are not modeled explicitly, as for the majority of operations within P2P simulations, these layers are not interesting. Therefore, the simulator contains four layers: the Network, the Transport, the Overlay, and the Application Layer. To allow for an easy interchangeability, each layer defines its own interfaces. An implementation has to conform to the interface or the existing skeletal implementations of its functional layer to enable its usage within the simulation scenarios. We name the appropriate interfaces and skeletal implementations at the beginning of the section of each layer. Later on, Chapter 5 details how configurations are made to set complex scenarios with these different layers up.

3.1 Network Layer

Within this section, we detail the lowest simulated layer of PeerfactSim.KOM by describing the general model that forms the basis of this layer, comprising the interfaces and the existing skeletal implementations. In addition, we describe the currently existing implementations in the following subsections, which can already be used for simulations.

The model that forms the basis of this layer comprises two components. On the one hand, it includes the simulated *Network Layer* at every Host, which exchanges messages with the Transport Layer above inside a Host (See Section 2.2.2 for more details). On the other hand, the model contains a further component called *Subnet*, which abstracts and represents the Internet. Therefore, the Network Layer of a Host is not just connected with the higher Transport Layer but also with the Subnet (displayed in Figure 3.1), whereby the Subnet connects all the simulated Hosts with each other and simulates the transmission of data between different Hosts through the Internet.

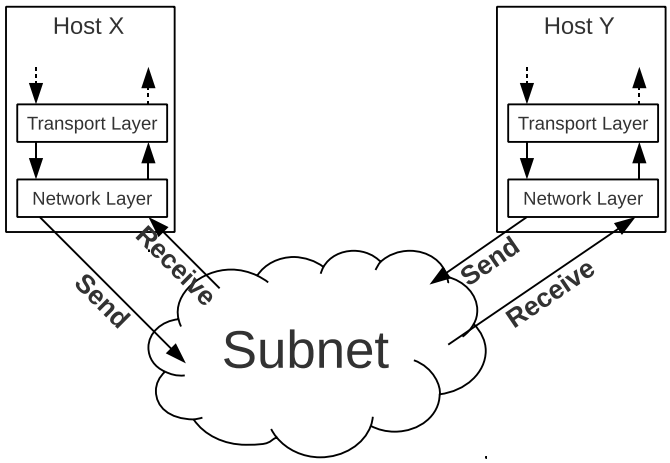


Figure 3.1: Communication between the different Network Layers of a simulation through the Subnet

Within a Host, the Network Layer is the only layer which is connected to the Subnet and therefore responsible to handle the state of this connection with the Subnet. So, when the Host is online, the connection between the layer and the Subnet is established, meaning that every higher layer inside the Host can exchange information via the Network Layer to the Subnet (e.g. the Overlay Layer can join the P2P overlay). Changes concerning the connection between the Network Layer and the Subnet are captured by the layer and reported to every component inside the Host, which is interested (see Subsection 2.2.1 for more details about this information service).

Taking the Subnet into consideration, it models the intrinsic complexity of the Internet as a *big cloud*, which appears to be transparent for the Hosts. That is, when sending a message, the message is forwarded from the sending Host to the the Subnet, which manages the calculation of transmission times, models the packet loss and jitter, and schedules the appropriate events at the simulation framework. It also triggers the arrival of a message at the appropriate receiving Host.

A concrete implementation of the Network Layer always comprises the implementation of the layer at the Host as well as of the Subnet so that a certain type of a Subnet can never be mixed with another type of Network Layer. The required skeletal implementations for the Network Layer, mentioned in the following, are situated in the `network`-package[1] while the related interfaces are provided by an other `network`-package[2].

To implement a complete new Network Layer, the new Subnet must extend and implement `AbstractSubnet`, while the concretion of the Network Layer at the Host should extend `AbstractNetLayer`. The latter abstract class provides a skeletal implementation of the `NetLayer`-interface to facilitate the implementation of this interface. The `NetLayer`-interface, in turn, provides the methods to communicate with the overlying Transport Layer as well as with the Subnet.

Considering the transmission of data, a message is pushed down to the Network Layer inside the sending Host. At the Network Layer, the transport message is embedded into a network message, which is passed to the Subnet. The Subnet forwards the message to the receiving Host and calculates delay, loss probability etc. as previously mentioned. The class that models a network message of the respective network model must extend the skeletal implementation `AbstractNetMessage`, which partially implements the `NetMessage`-interface. `NetMessage` defines the common functionality of a message at the network level and specifies, which kind of information such a message should contain (e.g. addresses of the sender and receiver, network protocol etc.).

After detailing the underlying concept of the Network Layer and giving insides about its functionality, its components (Network Layer at the Host and Subnet) and the provided skeletal implementations, the following subsections deal with the existing implementations of the Network Layer.

3.1.1 Simple Network Model

The *Simple Network Model*[3] is targeted to implement a stable and efficient network layer, without going too much into detail about network characteristics. It is mainly applicable on debugging and testing of overlays and provides a fast simulation run. Further aspects address high memory efficiency and a faster loading of network stacks at the beginning of the simulation. Currently, only User Datagram Protocol (UDP) network transmissions can be set up using this network layer. If not stated otherwise, the referenced classes and interfaces of the Simple Network Model can be found in `de.tud.kom.p2psim.impl.network.simple`.

[1] `de.tud.kom.p2psim.impl.network`
[2] `de.tud.kom.p2psim.api.network`
[3] `de.tud.kom.p2psim.impl.network.simple`

3.1.1.1 Concepts of the Simple Network Model

In the following, we detail the different concepts and aspects that were integrated in the contemplated network model.

Addressing

The applied addressing scheme of the Simple Network Model is based on Integers. The address of a Network Layer of a simulated Host is assigned through the corresponding factory `SimpleNetFactory`, starting with the ID 0, and increased by one for every new Network Layer.

Behavior

When the `SimpleNetLayer` of a Host sends a message to another Host, first, it is checked, whether the sender is online or offline. Messages from senders, which are offline, will be immediately dropped. If the sender is online, the corresponding Subnet `SimpleSubnet` of the network model calculates the latency for the transmission by scheduling the message delivery after a given time, which is determined by the chosen *latency model* (Subsection 3.1.1.2 details the available latency models for this network model). In order to ensure in-order delivery, the different strategies for the calculation of the latency ensure that previously sent messages will always be delivered before subsequently sent messages. When the delivery is executed at the receiving Host, it will be checked, whether the receiver is online, otherwise the packet will be dropped.

Message sizing

The size of the message `SimpleNetMessage` of the Simple Network Model is only the size of the carried payload. It does not add any packet header or footer size.

Bandwidth management

No bandwidth management for the message transmission at the Hosts is currently implemented in the Simple Network Model. The Hosts have virtually unlimited bandwidth. This means that neither the number of received or sent messages nor their size do matter. So, the latency will never increase and no packets will ever be dropped.

3.1.1.2 Latency Models

The Simple Network Model provides three models to define a latency for transmissions between a sender and receiver:

- Simple Latency Model
- Simple Static Latency Model
- Simple Variable Latency Model

Simple Latency Model

In the `SimpleLatencyModel` class, a random position is assigned to every Host on the two-dimensional surface of a torus with the size of $40.000 \cdot 40.000 km^2$. The formula for the calculation of the latency for a sender s and a receiver r is based on the idea of Kovacevic et al. [18] and defined as

$$latency(s,r) = f \cdot (df(r) + \frac{dist(s,r)}{v})$$

In this formula, $dist(s,r)$ denotes the shortest euclidean distance between the two Hosts s and r on the torus (see Figure 3.2), while $df(n)$ is an uniformly distributed time between 0 and 31ms, which is immutably bound to every Host n. The absolute term v represents the signal propagation speed of $100.000 km/s$ and f is a random number out of $(0.1, 0.2, ...1.0)$.

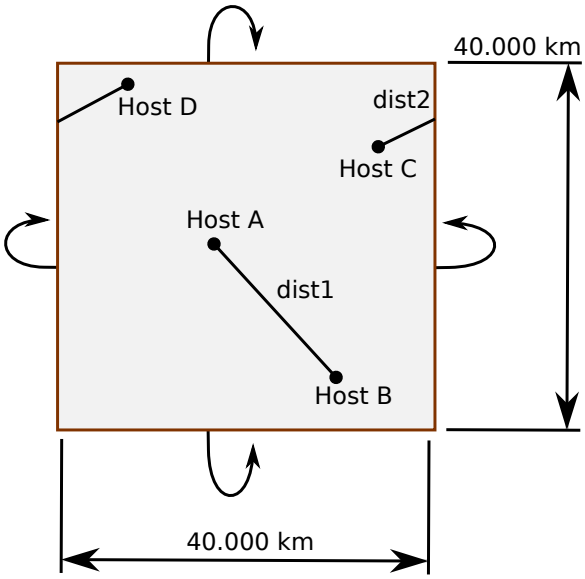


Figure 3.2: The euclidean distance between two pairs of Hosts on the two-dimensional surface of a torus

Simple Static Latency Model

This latency model, implemented in `SimpleStaticLatencyModel`, simply assigns one static latency to every transmission in the network. The default static latency is 10ms, but it can be changed via the *latency* parameter, as detailed in Listing 3.2.

Simple Variable Latency Model

This latency model consists of a static and a variable part and is implemented in `SimpleVariableLatencyModel`. The model requires two parameters, `base` and `variation`, which specify the values for the static and dynamic part of the latency and can be defined as explained in Listing 3.3. The latency is calculated through

$$latency(s,r) = base + variation \cdot f$$

where f is a uniformly distributed number out of $[-0.5, 0.5)$, which is calculated for every sent message.

3.1.1.3 Configuration

To use the provided model of Simple Network Layer, the corresponding factory has to be declared in the configuration file of a simulation, specifying the chosen latency model (further information on configuring a simulation is given in Section 5.1).

The following example defines the `SimpleNetworkLayerFactory` as Network Layer factory with `SimpleStaticLatencyModel` as latency model and a static latency of 10 milliseconds.

```
1 <Configuration>
2   [...]
3   <NetLayer class="de.tud.kom.p2psim.impl.network.simple.SimpleNetFactory">
4     <LatencyModel
          class="de.tud.kom.p2psim.impl.network.simple.SimpleStaticLatencyModel"
          latency="10ms"/>
5   </NetLayer>
6   [...]
7 </Configuration>
```

Listing 3.1: Declaration of the `SimpleNetworkLayerFactory`, using the `SimpleStaticLatencyModel`

The following example creates the SimpleNetworkLayerFactory as Network Layer factory with SimpleLatencyModel as latency model.

```
<Configuration>
  [...]
  <NetLayer class="de.tud.kom.p2psim.impl.network.simple.SimpleNetFactory">
    <LatencyModel
        class="de.tud.kom.p2psim.impl.network.simple.SimpleLatencyModel"/>
  </NetLayer>
  [...]
</Configuration>
```

Listing 3.2: Declaration of the SimpleNetworkLayerFactory with the SimpleLatencyModel

The following example creates the SimpleNetworkLayerFactory as Network Layer factory with SimpleVariableLatencyModel as latency model and a base latency of 30ms and a variying part of maximal 10ms.

```
<Configuration>
  [...]
  <NetLayer class="de.tud.kom.p2psim.impl.network.simple.SimpleNetFactory">
    <LatencyModel
        class="de.tud.kom.p2psim.impl.network.simple.SimpleVariableLatencyModel"
        base="30ms" variation="10ms"/>
  </NetLayer>
  [...]
</Configuration>
```

Listing 3.3: Declaration of the SimpleNetworkLayerFactory with a SimpleVariableLatencyModel with base set to 30*ms* and variation set to 10*ms*.

3.1.2 GNP Network Model

In contrast to the rather lightweight Simple Network Model, which is configurable by a few parameters and mainly consists of randomly generated values, the GNP Network Model, including the GNPNetLayer[4] and the GNPSubnet[5], provides a detailed simulation of network transmission characteristics based on empirical studies of the Round-Trip Time(RTT), jitter, geographical position, and packet loss of the today's Internet.

For this model, measurement data from the projects PingER[6], CAIDA[7], and MaxMind[8] was gathered, summarized and put into a single XML-file (the reference measurement file can be obtained from the PeerfactSim.KOM website[9]). In a compressed format, the file has a size of around 8MiB, while uncompressed, it is around 32MiB. The provided measurement file can be replaced with every other measurement data file for a better accuracy or to include recent measurements, as long as it conforms to the defined format, which is specified in Subsection 3.1.2.1.

The approach of the GNP Network Model to accurately model a world-scale computer network is described in detail by Kaune et al. [15] and Kunzmann et al. [21], while the implementation and integration into the simulator is detailed in [17]. In the following, a brief overview about the implemented model is given. First, the Hosts are placed in a hypothetical multidimensional euclidean space, as described in

4 de.tud.kom.p2psim.impl.network.gnp
5 de.tud.kom.p2psim.impl.network.gnp
6 http://www-iepm.slac.stanford.edu/pinger/
7 http://www.caida.org/home/
8 http://www.maxmind.com/app/ip-location
9 http://peerfact.kom.e-technik.tu-darmstadt.de/fileadmin/data/measured_data/measured_data.xml.zip

[27], so that their distances have a minimal error to their CAIDA minimum Round-Trip Time. This is an optimization problem, which can be solved by the *downhill-simplex algorithm* by Nelder and Mead [26]. The resulting positions are the measurement data source for the GNP Network Model, which can easily calculate an approximation of the Round-Trip Time between two peers based on their euclidean distance in this space. Second, a lognormal distributed jitter is synthesized by using the PingER measurements of Round-Trip Time deviation. Furthermore, geographical Host positions are defined using MaxMind's GeoIP database.

The GNP Network Model is very versatile regarding its data sources, for example, it is also possible to calculate the minimum Round-Trip Time based on the Host's geographical position (transmission speed).

3.1.2.1 Understanding the GNP Layer's data source

For a better understanding of the GNP Network Model, it is obviously helpful to understand its input. The measurement data is divided into four sections:

- The `Hosts` section lists Host addresses with geographical and GNP coordinates
- The `Group` section defines groups of Hosts above
- The `PingER` section contains Round-Trip Time parameters and packet loss rates, summarized to countries
- The `CountryLookup` section represents the relations between regions, countries and continents

Listing 3.4 shows the format of the Hosts section in the measurement file. Here, only two Hosts are defined as an example. In reality, this list may contain hundreds of thousands of example Hosts. The attributes of a Host element are:

- `ip`: the IPv4 address of the Host, represented by its four bytes concatenated to a 32-bit Integer
- `continentalArea`: the continent string, the Host belongs to
- `countryCode`: the 2-letter ISO 3166-1 alpha-2 country code of the country, the Host belongs to
- `region`: a region inside a country, the Host belongs to
- `isp`: an optional Internet Service Provider, the Host belongs to
- `latitude`: the geographical latitude of the Host in degrees, expressed as a `double`
- `longitude`: the geographical longitude of the Host in degrees, expressed as a `double`
- `coordinates`: the virtual multidimensional (here, 5-dimensional) GNP coordinates expressed as a comma-separated `double array`. The dimension count of the GNP coordinates can be varied, but one has to keep in mind that the dimension count must be the same on every Host in the file.

```
1 <gnp>
2   <Hosts>
3     <Host ip="1122978322" continentalArea="North America" countryCode="US"
          region="Texas" city="Dallas" isp="—" longitude="-96.79930114746094"
          latitude="32.80979919433594" coordinates="1422.7571119225017,
          1448.9159963309803, 1448.9159963309803, 1448.9159963309803,
          1448.9159963309803"/>
4     <Host ip="2527234561" continentalArea="Latin America" countryCode="BR"
          region="Santa Catarina" city="Florianopolis" isp="—"
          longitude="-48.56669616699219" latitude="-27.583297729492188"
          coordinates="1501.3574826182405, 1393.0879120975333, 1393.0879120975333,
          1393.0879120975333, 1393.0879120975333"/>
5     [...]
6   </Hosts>
```

Listing 3.4: The Hosts section of the GNP measurement file

The next section of the measurement file puts the Hosts above into several groups. These groups are commonly created for regions, countries, or continents. Every group element has an `id` attribute and a `maximum size` attribute representing the number of IP addresses it contains. Furthermore, it contains an element `IPs`, which has a `value` attribute that contains the IPv4 addresses of the Hosts that belong this group, again represented by its four bytes concatenated to a 32-bit Integer. An example is given in Listing 3.5.

```
1
2   <GroupLookup>
3     <Group id="Artvin" maxsize="4">
4      <IPs value="3254779137,3254779393,3254779649,3247242241"/>
5     </Group>
6     <Group id="Alagoas" maxsize="2">
7      <IPs value="3368507645,3372118529"/>
8     </Group>
9     [...]
10  </GroupLookup>
```

Listing 3.5: The group section of the GNP measurement file

In contrast to the sections above, the next section defines properties for *relations* of countries. The section is named after the PingER project, which provides the data source for these properties. These relations are an important source for the packet loss rate. The attributes for every relation are:

- `from`: the country of the sender
- `to`: the country of the receiver
- `minimumRtt`: the minimum Round-Trip Time that was measured
- `averageRtt`: the average Round-Trip Time that was measured
- `delayVariation`: the variation of the Round-Trip Time that was calculated from frequent measurements
- `packetLoss`: the packet loss rate that was discovered, given in percent

```
1   <PingErLookup>
2     <SummaryReport from="Korea, Rep" to="Portugal" minimumRtt="323.63"
          averageRtt="329.75" delayVariation="1.71" packetLoss="0.21"/>
3     <SummaryReport from="Korea, Rep" to="United States" minimumRtt="184.17"
          averageRtt="199.73" delayVariation="4.02" packetLoss="0.74"/>
4     [...]
5   </PingErLookup>
```

Listing 3.6: The PingEr section of the GNP measurement file

The last section is used to map regions to different countries and countries to their continents and country codes.

```
1   <CountryLookup>
2     <CountryKey code="LT" countryGeoIP="Lithuania" countryPingEr="Lithuania"
          regionPingEr="Europe"/>
3     <CountryKey code="EC" countryGeoIP="Ecuador" countryPingEr="Ecuador"
          regionPingEr="Latin America"/>
4     [...]
5   </CountryLookup>
6 </gnp>
```

Listing 3.7: Country mapping between different region types in the GNP measurement file

3.1.2.2 Behavior

The behavior of the GNP Network Model is roughly similar to the Simple Network Model (see Subsection 3.1.1), but the contemplated implementation of the network details much more aspects of the real-life Internet:

Addressing

Endpoints are addressed by their IPv4 network address[10].

Packet sizing

When speaking of packet sizing, we refer to the messages of the simulator, their sizes and how they are treated. The terms packet and messages can be used interchangeably. As far as possible, the packet sizing conforms to IPv4 and is implemented in `IPv4Message`[11]. A 20-byte header is added, the packet size is bound to 65535 bytes and the Maximum Transmission Unit (MTU) size is 1500 bytes, which is a silent agreement in the Internet.

Bandwidth management

In contrast to the Simple Network Model, the GNP Network Model simulates the maximum upstream and downstream bandwidth for the Hosts. If a message is currently processed by the endpoints (either being sent or received) and a further message needs to be handled, it is queued and processed when the packet before was handled. The time to process a packet is calculated by using the packet's size and the downstream/upstream bandwidth property of the Hosts.

Packet loss

If the parameter `usePingErPacketLoss` is set to true at the latency model in the configuration of the Network Layer (see Subsection 3.1.2.4 for details of the Network Layer configuration), the GNP Network Model simulates a packet loss for UDP transmissions, using the information in the PingEr section of the measurement file [12]. The loss rate for an UDP packet m from the sender s to the receiver v was specified in [17] and is defined as

$$UDPErrP(s,v,m) = 1 - (1 - IPLossP(s,r))^{frag(m)}$$

In the formular $frag(m)$ calculates the number of fragments, the UDP packet m is split into, while the probability for an IP packet loss is calculated through $IPLossP(s,r) = 1 - \sqrt{1 - PingErLoss(s,r)}$. $PingErLoss(s,r)$,in turn, is the loss probability of the *response* when pinging r from s. This information is taken from the PingER section by using the sender's and receiver's country for looking up the relation.

3.1.2.3 Latency Model

In contrast to the Simple Network Model, the GNP Network Model provides only one latency model, which is very heterogeneous. Since the measurement data allows to calculate latency in several ways, the GNP latency model can be configured with several parameters how to calculate latencies.[13]

Parameters

The GNP Network Model accepts the following parameters during its configuration (see Subsection 3.1.2.4 for details of the Network Layer configuration):

[10] `de.tud.kom.p2psim.impl.network.IPv4NetID`
[11] `de.tud.kom.p2psim.impl.network`
[12] `de.tud.kom.p2psim.impl.network.gnp.GnpLatencyModel`
[13] `de.tud.kom.p2psim.impl.network.gnp.GnpLatencyModel`

- `usePingErJitter`: Specifies, whether to calculate a jitter, based on the PingEr section of the measurement data
- `usePingErPacketLoss`: Specifies, whether to drop packets according to the packet loss rate given by the PingEr section of the measurement data
- `useAnalyticalRtt`: Specifies, whether to use the *haversine formula*[14] to calculate distances and delays, rather than using GNP distances
- `usePingErRttData`: Specifies, whether to use a Round-Trip Time calculated from the PingEr section than using the GNP distance

By default, these parameters are all set to `false`.

Latency

The latency of the GNP network layer, as specified by Kaune et al. [15] is defined as

$$latency = \frac{minRTT(s,r) + jitter(s,r)}{2}$$

where $minRTT(s,r)$ is the minimum Round-Trip Time and $jitter(s,r)$ is the *jitter*, the variation of the Round-Trip Time.

Minimum Round-Trip Time

The calculation of the Round-Trip Time depends on the configuration parameters. If `useAnalyticalRtt` and `usePingErRttData` are both set to `false` (default), the minimum Round-Trip Time is defined as

$$minRTT(s,r) = \sqrt{\sum_{i=0}^{d}(gnpCoords(s,i) - gnpCoords(r,i))^2}$$

where $gnpCoords(h,i)$ represents the GNP coordinate of Host h of dimension i and d is the number of dimensions used in the GNP coordinate space (in our reference file, $d = 5$). Basically, this is the euclidean distance between two Hosts in the GNP coordinate system.

If `useAnalyticalRtt` is set to true and `usePingErRttData` remains false, the minimum Round-Trip Time is defined as

$$minRTT(s,r) = 62ms + 0.02ms/km \cdot dist(s,r)$$

where $dist(s,r)$ is the geographical distance of two Hosts in km, using the *haversine formula*, see Figure 3.3.

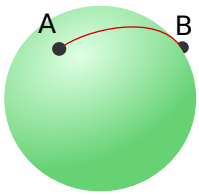


Figure 3.3: The haversine formula helps calculating distances on a sphere surface

If `usePingErRttData` is set to true, the minimum Round-Trip Time is simply taken from the PingEr project, while using the countries of the Host to find the relation.

[14] The *haversine formula* calculates distances *on* a sphere, like it has to be done between geographical coordinates on the earth surface

Jitter

Like the minimum Round-Trip Time, the calculation of the jitter depends on the configuration parameters. If `usePingErJitter` is set to `true`, the jitter is a random variable that is lognormal distributed. The parameters for the lognormal distribution, μ and σ, are calculated using an optimization algorithm [15] that takes the expected value and the variance as input. If `usePingErJitter` is false, no jitter is calculated at all.

3.1.2.4 Configuration

The configuration of the GNP Network Model does not significantly differ from the configuration of the Simple Network Model. The example in Listing 3.8 shows a very basic configuration of the network layer without any parameters for the latency model. A bandwidth is assigned to the Hosts according to the OECD broadband report (see Subsection 3.1.4.1 for details about the bandwidth assignment). The file with the GNP measurement data is located in `data/measured_data.xml`.

```
<Default>
  <Variable name="gnpDataFile" value="data/measured_data.xml" />
  [...]
</Default>
[...]
<NetLayer class="de.tud.kom.p2psim.impl.network.gnp.GnpNetLayerFactory"
  gnpFile="$gnpDataFile">
  <LatencyModel class="de.tud.kom.p2psim.impl.network.gnp.GnpLatencyModel" />
  <BandwidthDetermination
      class="de.tud.kom.p2psim.impl.network.bandwidthDetermination.
      OECDReportBandwidthDetermination" />
</NetLayer>
  [...]
```

Listing 3.8: Example of a GNP Network Model configuration

The example in Listing 3.9 shows the same configuration, but with all parameters of the latency model set to *true*. Additionally, a random bandwidth is assigned to Hosts with a maximum of 50KB/s and an upload bandwidth of 5KB/s (always 10% of the download bandwidth).

```
<Default>
  <Variable name="gnpDataFile" value="data/measured_data.xml" />
  [...]
</Default>
[...]
<NetLayer class="de.tud.kom.p2psim.impl.network.gnp.GnpNetLayerFactory"
  gnpFile="$gnpDataFile">
  <LatencyModel class="de.tud.kom.p2psim.impl.network.gnp.GnpLatencyModel"
      usePingErPacketLoss="true"
   useAnalyticalRtt="true" usePingErRttData="true" usePingErJitter="true" />
  <BandwidthDetermination
      class="de.tud.kom.p2psim.impl.network.bandwidthDetermination.
      RandomizedBandwidthDetermination"
    maxDownBandwith="50000" />
</NetLayer>
[...]
```

Listing 3.9: Example of a GNP Network Model configuration, with all latency model parameters set to `true` and a randomized bandwidth determination.

[15] Here, the Downhill-Simplex algorithm is used with 100 iterations

3.1.3 Modular Network Model

The many aspects of a network model that overlays and applications use during the simulation are versatile, but separable to a certain extent. This was the motivation for the development of the Modular Network Model, comprising the Network Layer `ModularNetLayer`[16] and its Subnet `ModularSubnet`[17], which consists of a model that is split up into multiple components modeling separable aspects of the network, called *strategies*.

The Modular Network Model is mainly based on the GNP Network Model, but the goal of the development of the modular network model is to provide a *flexible* configurability of the model (either simple or very detailed) and *comfortable* and *supported* extendability. Additionally, speed and memory consumption were a matter. The current version of the Modular Network Model currently supports the transmission of UDP, while the support for TCP has to be integrated in this model.

Modeling the delivery of a message from the sender to the receiver raises many problems beyond just applying a fixed latency. Here are the most important aspects that we will have to keep in mind:

- **Packet loss** occurs on the way from the sender to the receiver. This is mainly caused by intermediate routers, which drop packets when their queues are full, and by physical layer interference, especially on wireless media.
- Both sender and receiver often have a fixed **bandwidth**. The limited bandwidth is caused by the physical capacity of the last line to the subscriber, or by artificial traffic control restrictions: The Internet Service Provider (ISP) applies to its customers to avoid congestion of its network, caused by heavy-load users or malicious clients doing denial-of-service attacks to arbitrary targets in the Internet. Traffic control mechanisms modeling limited bandwidth will have to cause an additional latency for packet delivery, or even packet loss.
- When a message is too large to be transferred through the Internet, it will be split into multiple parts, which is called **fragmenting**. An appropriate network model must respect fragmentation when calculating e.g. packet loss.

In the following, we will explain how to configure the Modular Network Model (see Subsection 3.1.3.2) and how to include a measurement database (see Subsection 3.1.3.3). Then, we detail in Subsection 3.1.3.4 the available types of strategy modules for the Modular Network Model, including their default strategy module implementations that are shipped with the Modular Network Model and how to configure them.

3.1.3.1 Architecture

The core architecture (see Figure 3.4) is inherited from the abstract network layer architecture, comprising `AbstractNetLayer`[18] and `AbstractSubnet` [19]. While the Modular Network Model is a concretion of the Network Layer of PeerfactSim.KOM, the model can be additionally considered as a framework for different *strategy components* that define the model for the Network Layer. Strategy components, configured in `ModularNetLayerFactory`[20], are encapsulated and can be accessed via the interface `IStrategies`[21]. Then, the strategies can be utilized by the Hosts through the `ModularNetLayer` and by the modeled network through the `ModularSubnet`.

[16] `de.tud.kom.p2psim.impl.network.modular`
[17] `de.tud.kom.p2psim.impl.network.modular`
[18] `de.tud.kom.p2psim.impl.network`
[19] `de.tud.kom.p2psim.impl.network`
[20] `de.tud.kom.p2psim.impl.network.modular`
[21] `de.tud.kom.p2psim.impl.network.modular`

An important component of the Modular Network Model is the measurement database that is used (`NetMeasurementDB`[22]). The `NetMeasurementDB` extends the `RelationalDB`[23], which implements a simple relational schema. The tables of the database are declared using inner classes of `RelationalDB`.

The Modular Network Model uses `ModularNetMessage`[24] as message type at the Network Layer that should not be altered. The properties of `ModularNetMessage`, such as packet/header size or fragment count, are specified by the respective strategy components.

The skeletal implementations of a strategy are placed in the subpackage `.st`[25] with different subpackages for every strategy type containing the default strategy implementations as listed in Table 3.5. Customized strategies may also be placed here if they are useful for other projects, or otherwise at a customized package of the respective component.

3.1.3.2 Configuration

The Modular Network Model is targeted to be flexible regarding its configuration. One can choose to use presets of modules if one does not want to mess around with the detailed configurations. These presets choose a set of recommended strategies. To use a preset, the attribute `preset="Fundamental"` must simply be added to the NetLayer element in the XML configuration (see Listing 3.10). Here, the attribute for the preset has the value `Fundamental`.

```
1 [...]
2 <NetLayer class="de.tud.kom.p2psim.impl.network.modular.ModularNetLayerFactory"
    preset="Fundamental">
3   <BandwidthDetermination
      class="de.tud.kom.p2psim.impl.network.bandwidthDetermination.-
4      OECDReportBandwidthDetermination"/>
5 </NetLayer>
6 [...]
```

Listing 3.10: Example of a Modular Network Model configuration, setting the preset “Fundamental”.

An overview about modules and presets that are available can be found in Figure 3.6. After setting a preset, the Modular Network Model configuration can incrementally be altered by assigning specialized strategies. Strategy modules are configured by placing XML elements inside the NetLayer element. A strategy module of every strategy type can be placed inside the NetLayer element. Currently allowed types are *Packet Sizing*, *Fragmenting*, *Traffic Control*, *Packet Loss*, *Latency*, *Jitter* and *Positioning*.

Let us assume that we want to use the module `InfiniteTrafficQueue`[26] instead of `BoundedTrafficQueue`[27]. Then, we just declare this strategy module in the configuration, as displayed in the following listing.

Parameter `useRegionGroups`

This is a boolean parameter, which is per default set to `false`. This parameter is only required if a measurement database is loaded. If it is set to `false`, the Modular Network Model will suppose that the groups are not named by regions. It will randomly assign a Host metadata set (IP, Coordinates etc.) from the database, i.e. of a random geographical region. As a result, groups can have arbitrary names. If set to `true`, the Modular Network Model behaves exactly like the GNP Network Model when creating Hosts that are defined as groups in the XML configuration file. It will look in the groups section of the measurement

[22] `de.tud.kom.p2psim.impl.network.modular.db`
[23] `de.tud.kom.p2psim.impl.util.db.relational`
[24] `de.tud.kom.p2psim.impl.network.modular`
[25] `de.tud.kom.p2psim.impl.network.modular.st`
[26] `de.tud.kom.p2psim.impl.network.modular.st.trafCtrl`
[27] `de.tud.kom.p2psim.impl.network.modular.st.trafCtrl`

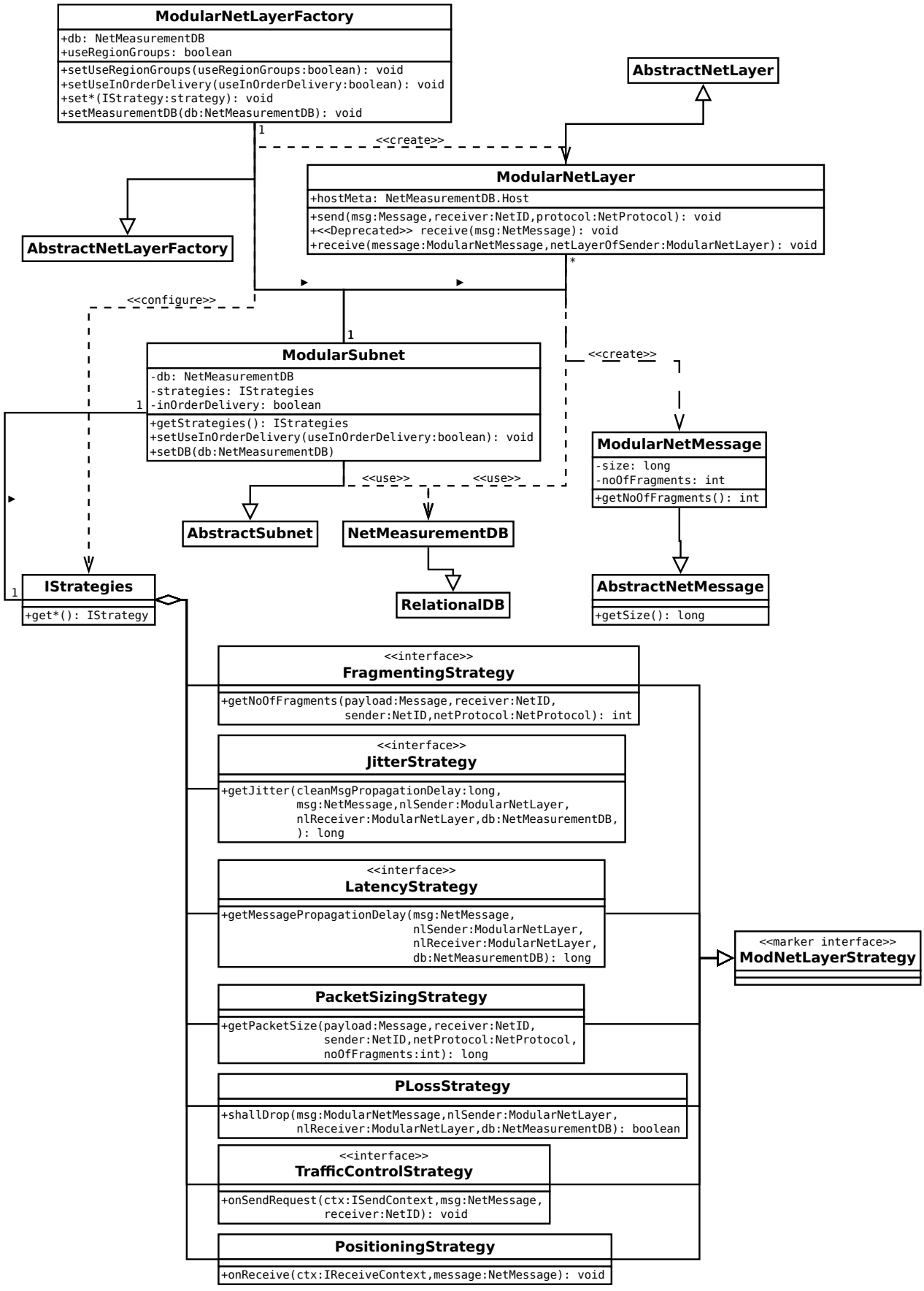


Figure 3.4: Excerpt of the Modular Network Model architecture

database for the assigned group name and pick a Host metadata set from this group. Usually, groups are required to have a region, country or continent name. If the group with the name could not be found in the database, an error is thrown.

```
[...]
<NetLayer class="de.tud.kom.p2psim.impl.network.modular.ModularNetLayerFactory"
 preset="Fundamental">
  <BandwidthDetermination
      class="de.tud.kom.p2psim.impl.network.bandwidthDetermination.-
       OECDReportBandwidthDetermination"/>
  <TrafficControl class="de.tud.kom.p2psim.impl.network.modular.st.trafCtrl.-
       InfiniteTrafficQueue"/>
</NetLayer>
[...]
```

Listing 3.11: Example of a Modular Network Model configuration, setting the preset `Fundamental` and incrementally changing the *Traffic Control* strategy to `InfiniteTrafficQueue`.

Parameter `inOrderDelivery`

This is a boolean parameter, which is set to `false` per default. This ensures that all packets will be delivered in the same order, in which they were sent (ignoring dropped packets). Sometimes, this is useful to be set when debugging, but one should keep in mind that an out-of-order delivery can be the typical behavior of a network.

3.1.3.3 Including and using measurements from a database

Some strategies of the Modular Network Model require a measurement database to work properly. If no database is loaded, these strategies will throw an error. To load a database, the XML element, called `MeasurementDB`, must be included inside the `NetLayer` element. The default database for the contemplated network model, which contains the relevant measurement data for the strategies can be downloaded at [28]. Listing 3.12 shows the configuration of the measurement database.

```
[...]
<NetLayer class="de.tud.kom.p2psim.impl.network.modular.ModularNetLayerFactory"
    [...]>
  [..]
  <MeasurementDB
      class="de.tud.kom.p2psim.impl.network.modular.db.NetMeasurementDB"
      file="data/mod_measured_data.xml"/>
  [..]
</NetLayer>
[...]
```

Listing 3.12: Including the default network measurement database scheme, supplying all strategies of the Modular Network Model with the required measurement data. The measurement data is read from the file in the attribute `file`

3.1.3.4 The offered strategy types and their implementations

The Modular Network Model is shipped with a default set of strategy modules (see Figure 3.5). Users of PeerfactSim.KOM are encouraged to develop additional strategies. In this section, we will introduce these strategy modules.

[28] https://dev.kom.e-technik.tu-darmstadt.de/redmine/attachments/download/848/mod_measured_data.xml.zip

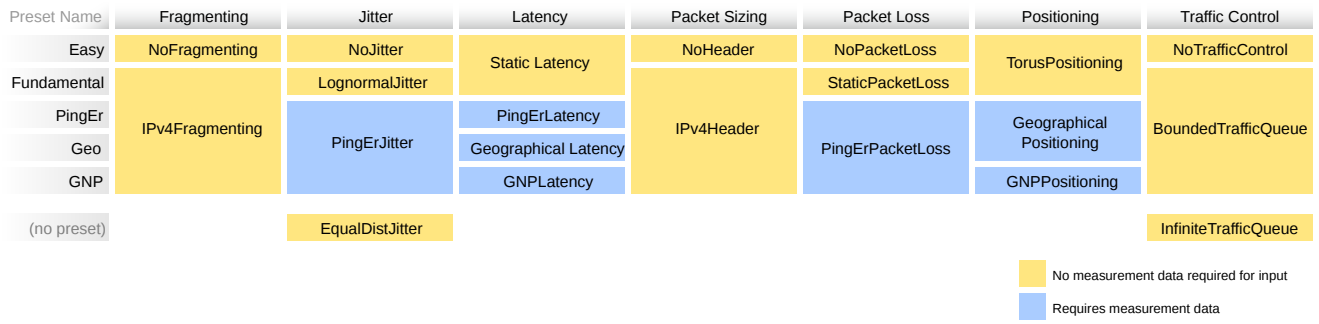


Figure 3.5: Overview of strategy components that are shipped with the Modular Network Model, grouped into different strategy types (columns) and presets (rows)

3.1.3.5 Fragmenting strategies

A network layer usually splits the Service Data Unit (SDU) up into multiple fragments, if it is too big. This strategy determines how many fragments are needed to transmit the given payload.

NoFragmenting

No fragmenting is used at all. This strategy will never split up messages, no matter how large they may get.

```
1 <Fragmenting
    class="de.tud.kom.p2psim.impl.network.modular.st.fragmenting.NoFragmenting"/>
```

Listing 3.13: Configuration of `NoFragmenting`

IPv4Fragmenting

This strategy applies the fragmenting strategy used by IPv4[29] with a supposed MTU of 1500 bytes. In IPv4 the number of fragments n is defined as

$$n = \lceil \frac{l}{MTU - h} \rceil$$

where l is the length of the payload and h the length of the IPv4 header (both in bytes). h is subtracted, since every fragment needs an own IPv4 header.

```
1 <Fragmenting
    class="de.tud.kom.p2psim.impl.network.modular.st.fragmenting.IPv4Fragmenting"/>
```

Listing 3.14: Configuration of `IPv4Fragmenting`

3.1.3.6 Jitter strategies

Jitter is the variation of the message propagation delay in a network. In the Modular Network Model, the total time that a message is delayed (see Figure 3.6) is defined as the message propagation delay (defined by the Latency strategy) plus the jitter (defined by the Jitter strategy), plus the time the message stays in sender and receiver queues defined by the Traffic Control strategy (if the traffic control mechanism delays the messages at all).

[29] Internet Protocol version 4

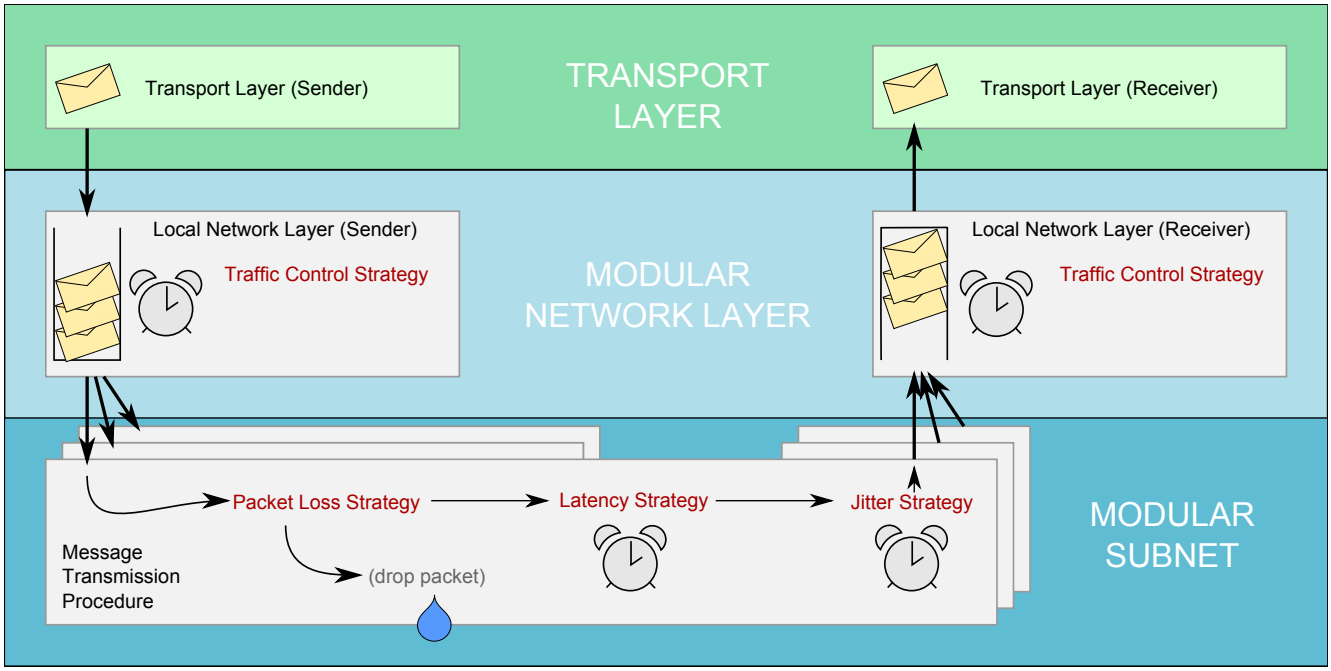


Figure 3.6: Strategies influencing the total delay and packet loss behavior of the Modular Network Model of a message that is sent from a sender to a receiver

NoJitter

This implementation applies no jitter at all. The delay between two Hosts will not vary.

```
1 <Jitter class="de.tud.kom.p2psim.impl.network.modular.st.jitter.NoJitter"/>
```

Listing 3.15: Configuration of `NoJitter`

EqualDistJitter

The jitter that is applied is a random value equally distributed between 0 and max. `max` is a parameter of `EqualDistJitter` and accepts simulation time units. By default `max="5ms"`.

```
1 <Jitter class="de.tud.kom.p2psim.impl.network.modular.st.jitter.EqualDistJitter"
    max="5ms"/>
```

Listing 3.16: Example configuration of `EqualDistJitter`

LognormalJitter

The jitter is lognormal distributed. The parameters of the lognormal function are μ and σ, The corresponding parameters of `LogNormalJitter` are denoted as `mu` and `sigma`. By default, `mu="1"` and `sigma="0.5"`. The units are milliseconds.

```
1 <Jitter class="de.tud.kom.p2psim.impl.network.modular.st.jitter.LognormalJitter"
    mu="1" sigma="0.5"/>
```

Listing 3.17: Example configuration of `LognormalJitter`

PingErJitter

The jitter is lognormal distributed, parameters are taken for every pair of Hosts from the PingER project. This strategy requires the measurement database, which contains the PingER project's measurement data. Since the PingEr project only contains values for the minimum Round-Trip Time and delay variation, the parameters μ and σ are approximated using the Downhill-Simplex optimization algorithm.

```
<Jitter class="de.tud.kom.p2psim.impl.network.modular.st.jitter.PingErJitter"/>
```

Listing 3.18: Configuration of `PingErJitter`

3.1.3.7 Latency strategies

Latency strategies define the message propagation delay of a given network message. This is the total time the network message is delayed minus the jitter, minus the time the message stays in sender and receiver queues defined by the traffic control strategy (if the traffic control mechanism delays the messages at all).

StaticLatency

This implementation returns a fixed propagation delay. The corresponding parameter of `StaticLatency` is denoted as `propagationDelay` and accepts simulation time units. By default, `propagationDelay="10ms"`.

```
<Latency
    class="de.tud.kom.p2psim.impl.network.modular.st.latency.StaticLatency"
    propagationDelay="10ms"/>
```

Listing 3.19: Example configuration of `StaticLatency`

PingErLatency

The minimum Round-Trip Time between two Hosts from the PingER project is taken as the message propagation delay. This strategy, requires the measurement database, which contains the PingER project's measurement data.

```
<Latency
    class="de.tud.kom.p2psim.impl.network.modular.st.latency.PingErLatency"/>
```

Listing 3.20: Configuration of `PingErLatency`

GeographicalLatency

Considers the geographical distance between two Hosts to calculate the latency that is applied. Like an option in the GNP Network Model, the propagation delay between two Hosts s and r is defined as

$$propagationDelay(s,r) = 62ms + 0.02ms/km \cdot dist(s,r)$$

where $dist(s,r)$ calculates the geographical distance of two Hosts in km, using the haversine formula (see Figure 3.3). This strategy requires the measurement database, which contains the geographical position of every Host.

```
<Latency class="de.tud.kom.p2psim.impl.network.modular.st.latency.-
    GeographicalLatency"/>
```

Listing 3.21: Configuration of `GeographicalLatency`

GNPLatency

This strategy calculates the latency through the euclidean distance between two Hosts in the GNP space, similar to the calculation in the GNP Network Model as explained in Subsection 3.1.2.3. This strategy requires the measurement database, which contains the GNP position of every Host.

```
<Latency class="de.tud.kom.p2psim.impl.network.modular.st.latency.GNPLatency"/>
```

Listing 3.22: Configuration of `GNPLatency`

3.1.3.8 Packet Sizing strategies

These strategies define how the size of network message calculated. This may include the size of the network message itself, as well as the sum of the size of all fragments of its carried *payload*.

NoHeader

The network messages do not have a header, are not padded or contain an error correction code that enlarges them. They are simply the size of the Transport Layer payload they have to carry.

```
1  <PacketSizing
       class="de.tud.kom.p2psim.impl.network.modular.st.packetSizing.NoHeader"/>
```

Listing 3.23: Configuration of `NoHeader`

IPv4Header

An IPv4 header of 20 bytes is added to the payload of the message.

```
1  <PacketSizing
       class="de.tud.kom.p2psim.impl.network.modular.st.packetSizing.IPv4Header"/>
```

Listing 3.24: Configuration of `IPv4Header`

3.1.3.9 Packet Loss strategies

Packet Loss strategies determine whether a given packet shall be dropped or not. Packet loss at the Network Layer can have many causes, like interference on the physical layer or congestion in intermediate routers, which cause them to drop subsequent packets.

NoPacketLoss

The Packet Loss strategy propagates all packets to the receiver without applying loss. Note that even when no packet loss is set, packets may be dropped in the sender and receiver traffic control, depending on the applied Traffic Control strategy.

```
1  <PLoss class="de.tud.kom.p2psim.impl.network.modular.st.ploss.NoPacketLoss"/>
```

Listing 3.25: Configuration of `NoPacketLoss`

StaticPacketLoss

Packets are dropped with a static probability. The corresponding parameter of `StaticPacketLoss` is denoted as `ratio` and accepts a double value from 0 to 1. By default, `ratio="0.005"`

```
1  <PLoss class="de.tud.kom.p2psim.impl.network.modular.st.ploss.StaticPacketLoss"
       ratio="0.005"/>
```

Listing 3.26: Example configuration of `StaticPacketLoss`

PingErPacketLoss

Packets are dropped with a static probability, but the probability differs between every sender and receiver. The corresponding ratio is taken from the PingER project. This strategy requires the measurement database, which contains the PingER project's measurement data.

```
<PLoss
    class="de.tud.kom.p2psim.impl.network.modular.st.ploss.PingErPacketLoss"/>
```

Listing 3.27: Configuration of `PingErPacketLoss`

3.1.3.10 Positioning strategies

Positioning strategies determine an abstract network position of a given Host, as well as the distances between Hosts.

TorusPositioning

This implementation applies a uniform distributed position on a 2- or multidimensional torus surface, like it is done in the Simple Network Model (we refer to Subsection 3.1.1.2). The distance between two Hosts is defined as the shortest euclidean distance on the surface.

The torus has the same size for every dimension of its surface (square, cube, hypercube). The corresponding parameters of `TorusPositioning` are written `torusDimensionSize` and `noOfDimensions`. By default, `torusDimensionSize="1"` and `noOfDimensions="2"`

```
<Positioning
    class="de.tud.kom.p2psim.impl.network.modular.st.positioning.TorusPositioning"
    torusDimensionSize="1" noOfDimensions="2"/>
```

Listing 3.28: Example configuration of `TorusPositioning`

GeographicalPositioning

The geographical position of the Hosts is used. The distance between two Hosts is defined as their shortest geographical distance on the earth surface This distance is calculated using the haversine formula explained in Subsection 3.1.2.3.

```
<Positioning class="de.tud.kom.p2psim.impl.network.modular.st.positioning.-
    GeographicalPositioning"/>
```

Listing 3.29: Configuration of `GeographicalPositioning`

GNPPositioning

The position of the Hosts is embedded in the abstract multidimensional GNP space (see Section 3.1.2 for details). The distance between two Hosts, i.e. their supposed latency, is the euclidean distance in this space, as described by [27].

```
<Positioning
    class="de.tud.kom.p2psim.impl.network.modular.st.positioning.GNPPositioning"/>
```

Listing 3.30: Configuration of `GNPPositioning`

3.1.3.11 Traffic Control strategies

If we would simply delay every message that is being sent and received, every peer would be able to send and receive an unlimited amount of data in any time interval, not taking care about available bandwidth. This is inappropriate for a realistic network layer behavior. In the Network Layer of PeerfactSim.KOM, a bandwidth is assigned to every Host. We can use this bandwidth to control the incoming and outgoing traffic of every peer, using different strategies.

NoTrafficControl

No traffic control is applied at all, the Hosts have virtually unlimited bandwidth.

```
<TrafficControl
    class="de.tud.kom.p2psim.impl.network.modular.st.trafCtrl.NoTrafficControl"/>
```

Listing 3.31: Configuration of `NoTrafficControl`

InfiniteTrafficQueue

This strategy implements a traffic queue that queues messages, if the sender attempts to send too fast, or if the receiver receives too many messages, regarding the bandwidth that is available for the current Host. This strategy may increase the total latency of the message delivery. The size of the queue is unlimited. The advantage is that no message ever gets dropped by this strategy, but it may cause large message latencies and a high memory consumption during a simulation.

```
<TrafficControl class="de.tud.kom.p2psim.impl.network.modular.st.trafCtrl.-
    InfiniteTrafficQueue"/>
```

Listing 3.32: Configuration of `InfiniteTrafficQueue`

BoundedTrafficQueue

This strategy is similar to the `InfiniteTrafficQueue`, but has a fixed queue size. Queue sizes are given in simulation time units, meaning that the current message shall be dropped on arrival, if the *last* message that was queued still would have to wait longer than the queue size period.

The corresponding queue size parameters of `BoundedTrafficQueue` are denoted as `maxTimeSend` and `maxTimeReceive`, both in simulation time units. By default, `maxTimeSend="3s"` and `maxTimeReceive="3ms"`, as well.

```
<TrafficControl
    class="de.tud.kom.p2psim.impl.network.modular.st.trafCtrl.BoundedTrafficQueue"
    maxTimeSend="3s" maxTimeReceive="3s"/>
```

Listing 3.33: Example configuration of `BoundedTrafficQueue`

3.1.4 Bandwidth determination

In the Network Layer, every Host needs a *bandwidth*. The implementations of the Network Layer do not give a realistic bandwidth determination model themselves.Thus, separate bandwidth determination strategies can be defined to assign an up- and downstream bandwidth to every Host.

By default, the up- and downstream bandwidth of a Host are set to 5 kilobytes per second as defined in `AbstractNetLayerFactory`[30]. This fixed bandwidth can be altered by explicitly stating new values in the declaration of the Network Layer definition (see listing 3.34).

```
[...]
<NetLayer class="[...]" downBandwidth="10000" upBandwidth="2000">
  [...]
</NetLayer>
[...]
```

Listing 3.34: Example of assigning a fixed bandwidth of 10KB/s downstream and 2KB/s upstream to every Host.

However, more advanced mechanisms allow a versatile assignment of bandwidths to Hosts.

[30] `de.tud.kom.p2psim.impl.network`

3.1.4.1 OECD-Report-based

The Organisation for Economic Co-operation and Development (OECD) publishes a broadband report on an annual basis[31]. The following table is based on a report of the utilized network access technologies in OECD countries that has been published in June 2009. The frequency of modem and ISDN users are not from this report and are estimated.

Type	Up (kbps)	Down (kbps)	Up (Byte/s)	Down (Byte/s)	Absolute freq.	Rel. freq.
Wireless	712	1 889	89000	236111	4.703.298	0,012
DSL	699	9 623	87355	1202843	160.194.693	0,414
Cable	1 264	14 856	158001	1856966	75.807.109	0,196
FTTx	34 247	65 326	4280818	8165793	25.834.733	0,067
Modem	56	56	7000	7000	60.000.000	0,155
ISDN	64	64	8000	8000	60.000.000	0,155

Whenever a bandwidth is assigned to a Host, a bandwidth is randomly chosen from one of these table rows, weighted with the according frequency. In large scale simulations, this will approximate the distribution of the OECD broadband report.

The OECD-Report-based bandwidth assignment must be declared in the configuration of the Network Layer (see Listing 3.35).

```
1 [...]
2 <NetLayer class="[...]">
3   [...]
4   <BandwidthDetermination
        class="de.tud.kom.p2psim.impl.network.bandwidthDetermination.
        OECDReportBandwidthDetermination" />
5 </NetLayer>
6 [...]
```

Listing 3.35: Example of declaring the OECD-Report-based bandwidth determination.

[31] http://www.oecd.org/sti/ict/broadband

3.2 Transport Layer

Within this section, we describe the Transport Layer of PeerfactSim.KOM. This includes a brief introduction to the layer's general responsibilities, as well as a detailed description of its implementation and configuration.

The Transport Layer's main task is to provide an end-to-end communication service to higher layers, like P2P overlays or applications. As they may have differing requirements for this communication, there are several optional services the Transport Layer may provide. This includes services, such as multiplexing by using the notion of ports, connection-oriented data streams, error-correction, and flow-control [30]. For simulations the question if, and in what details, these services are realized, depends on the simulation's purpose. As PeerfactSim.KOM was primarily built to simulate large-scale P2P systems, the detailed mechanisms provided by the Transport Layer are most of the time not a focus of simulations. The implementation of this layer therefore may abstract over the correct implementation of most services to allow efficient and still valuable simulations of higher layers.

3.2.1 Transport Layer interfaces and abstract classes

For the implementation of Transport Layers, PeerfactSim.KOM provides some basic interfaces and abstract classes to be implemented, plus some standard classes to be used. The interfaces are located in the package `transport`[32], the abstract classes and standard classes in the package `transport`[33]. They are shown in Figure 3.7 and described in the following paragraphs.

`TransInfo`[34]: This interface defines a Transport Layer address to consist of a Network Layer address bundled with a port number to allow multiplexing of multiple transport connections via a single network connection.

`TransMessageEvent`[35]: This class is used as a container to pass information about a received Transport Layer message to registered listeners (see the next paragraph about the `TransMessageListener` interface). It includes the `TransInfo` of the sender, the used `TransProtocol`, the communication Id, and the payload of the received Transport Layer message.

`TransMessageListener`[36]: This interface has to be implemented to register a class as event handler for incoming Transport Layer messages at a given port. The registration/de-registration is done via the `TransLayer` methods `addTransMsgListener(TransLayerListener,short)` and `removeTransMsgListener(TransLayerListener,short)`. After a registration, the `TransMessageListener` is notified on arrival of new messages at the stated port. The notification is done by a call of the method `messageArrived(TransMsgEvent)` and the passing of an instance of `TransMsgEvent`. This instance includes all relevant information of the received message, including the payload message itself.

The concept of registering listeners for certain events, follows the idea of the Observer Design Pattern [10]. It is applied at all layers of the simulator and used to allow an efficient communication between layers inside a Host. Subsection 2.2.2 gives an overview about this mechanism and its importance in the layered architecture of PeerfactSim.KOM.

`TransMessageCallback`[37]: This interface has to be implemented to wait for and handle replies when using the `TransLayer` method `sendAndWait`. This is different from waiting for new incoming connection

[32] `de.tud.kom.p2psim.api.transport`
[33] `de.tud.kom.p2psim.impl.transport`
[34] `de.tud.kom.p2psim.api.transport.TransInfo`
[35] `de.tud.kom.p2psim.impl.transport.TransMessageEvent`
[36] `de.tud.kom.p2psim.api.transport.TransMessageListener`
[37] `de.tud.kom.p2psim.api.transport.TransMessageCallback`

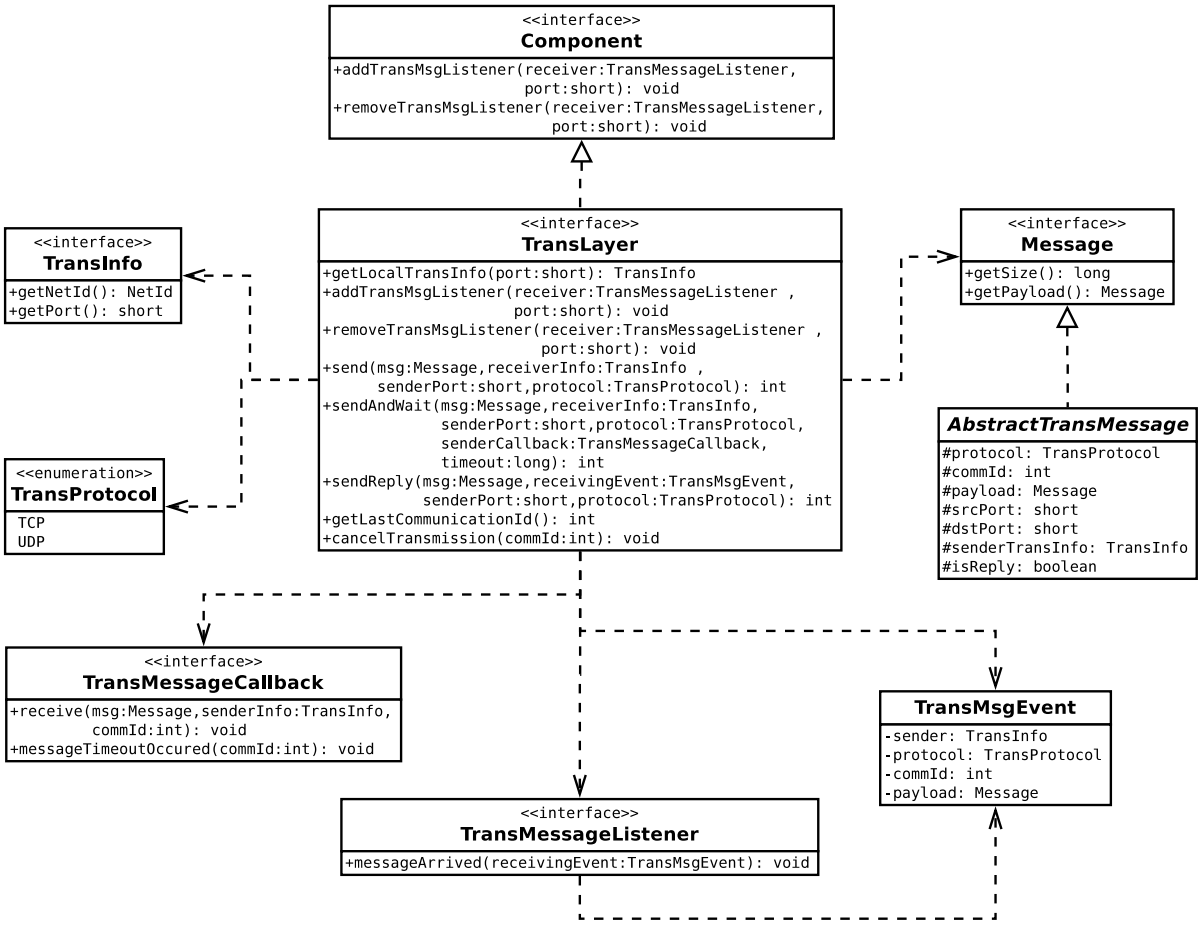


Figure 3.7: The Transport Layer API

requests which in contrast uses the interface `TransMessageListener`. A `TransMessageCallback` waits for incoming replies passed from the `TransLayer` and can perform actions when a reply is received. More information on `sendAndWait` and its idea can be found in Subsection 3.2.2.

`TransProtocol`[38]**:** This is the enumeration which defines the basic types of Transport Layer messages the simulator supports. As each implementation of a transport message should inherit from the abstract class `AbstractTransMessage`, it automatically holds a field to state the used transport protocol type defined by this enumeration. The field should always be set in the constructor of a transport message. It is used at various points and has influence on the treatment of a message.

So far, we only distinguish between the protocols TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP is defined as being connection oriented and reliable, while UDP is defined as being connectionless and unreliable. These properties are part of the enumeration's definition and can be obtained from each element by the getter-methods `isConnectionOriented()` and `isReliable()`.

`TransLayer`[39]**:** This is the most central interface for a Transport Layer implementation. It defines the complete interface for the Transport Layer's services and has to be implemented to integrate its implementation into the simulator. Classes that implement this interface have the following two main responsibilities.

[38] `de.tud.kom.p2psim.api.transport.TransProtocol`
[39] `de.tud.kom.p2psim.api.transport.Translayer`

The first is to accept messages of higher layers and to transmit them to another Host using the underlying Network Layer. To do so, the interface defines the method `send(Message,TransInfo,short,TransProtocol)`. The methods `senAndWait(Message,TransInfo,short,TransProtocol,TransMessageCallback,long)` and `sendReply(Message,TransMsgEvent,short,TransProtocol)` are also used for message transmission. Their purpose is to allow higher layers an easy implementation of request-reply scenarios. Further information about this feature can be found in Subsection 3.2.2.

The second responsibility is to handle incoming messages that the Network Layer passes to the Transport Layer and to dispatch them to local higher layer processes. For the second task, the class has to maintain a mapping between local port numbers and processes that have registered for a specific port. To allow processes to register and de-register for a specific port, the interface defines the two methods `addTransMsgListener(TransMessageListener,short)` and `removeTransMsgListener(TransMessageListener,short)`.

`AbstractTransMessage`[40]**:** This is the base class for the implementation of all Transport Layer messages. It defines the fields common to all transport messages, as well as their required getter and setter methods.

`TransMsgEvent`[41]**:** This class encapsulates all data about a received Transport Layer message. It is used to pass this information, including the message itself, to the registered `TransMessageListeners`. They can use passed instances of this class to obtain the included information and for answering messages that where send using the `sendAndWait` mechanism. The latter is described in Subsection 3.2.2.

3.2.2 The `sendAndWait` concepts

Beside the method `send(Message,TransInfo,short,TransProtocol)` for basic message sending, the interface `TransLayer` includes the method `sendAndWait(Message,TransInfo,short,TransProtocol,TransMessageCallback,long)`. This method realizes a functionality which traditionally is not part of the Transport Layer. It was introduced to the simulator's Transport Layer to avoid overlays or applications implementing the mechanism multiple times.

The basic idea is to give a receiver the possibility to directly answer an incoming request with a reply message. The mapping between request and reply is done by the Transport Layer and does not have to be implemented at overlay or application level. The sender provides an instance of `TransMessageCallback` when sending a message, which is then used to notify the sender about a reply to its request. Beside this, the callback is used to notify the sender about an exceeded timeout. This timeout is specified by the caller when sending the request message.

Listing 3.36 shows a simple example how `sendAndWait` could be used at sender side. This source code could, for example, occur in any method of an `OverlayNode` implementation. It shows the instantiation of a `TransMessageCallback` and the point where to add own logic to be executed on reply arrival or exceeding of the timeout. In this example we specify UDP as transport protocol (line 18) and set the timeout to 20 simulated seconds (line 20).

At receiver side, there is an easy way to answer requests that were send using the `sendAndWait` method. Listing 3.37 shows an example for the sending of a reply message which, at sender side, is automatically mapped to the request and results in the call of the provided `TransMessageCallback`. Typically, the receiver implements this functionality in the class that implements the interface `TransMessageListener` and handles incoming messages. To answer the received request, it uses the

[40] `de.tud.kom.p2psim.impl.transport.AbstractTransMessage`
[41] `de.tud.kom.p2psim.impl.transport.TransMsgEvent`

method `sendReply(Message,TransMsgEvent,short,TransProtocol)` of the Host's Transport Layer.

```
TransMessageCallback senderCallback = new TransMessageCallback() {

  @Override
  public void receive(Message msg, TransInfo senderInfo, int commId) {
    // Do something on arrival of reply message
  }

  @Override
  public void messageTimeoutOccured(int commId) {
    // Do something when timeout is exceeded
  }
};

getTransLayer().sendAndWait(
          someRequestMsg,               // The Message to be send
          receiverInfo,                 // The TransInfo of the receiver
          senderPort,                   // The port of the sender
          TransProtocol.UDP,            // The Transport Layer protocol to be used
          senderCallback,               // The callback, used when reply arrives
          20 * Simulator.SECOND_UNIT);  // The timeout interval
```

Listing 3.36: Simple example sending a request using `sendAndWait`

```
[...]
  @Override
  public void messageArrived(TransMsgEvent receivingEvent) {
    [...]

    // The message that is send as reply
    SomeMessageType replyMsg = new SomeMessageType();

    node.getTransLayer().sendReply(
                replyMsg,               // The reply message
                receivingEvent,         // The receiving event of the request
                senderPort,             // The port of the sender
                TransProtocol.UDP);     // The Transport Layer
    [...]
  }
[...]
```

Listing 3.37: Simple example answering a request using `sendAndWait`

3.2.3 Default Transport Layer model

So far, this is the only implementation of a Transport Layer for PeerfactSim.KOM. All of its classes are located in the package `transport`[42]. In the following subsection, we will give an overview and describe the classes in detail.

[42] `de.tud.kom.p2psim.impl.transport`

3.2.3.1 Classes

Figure 3.8 gives an overview of the classes involved in the default Transport Layer implementation and its dependencies to the aforementioned API. To keep it simple, the class diagram does not include all interface methods and dependencies between the API elements themselves. We refer to Figure 3.7 for a detailed diagram of the API.

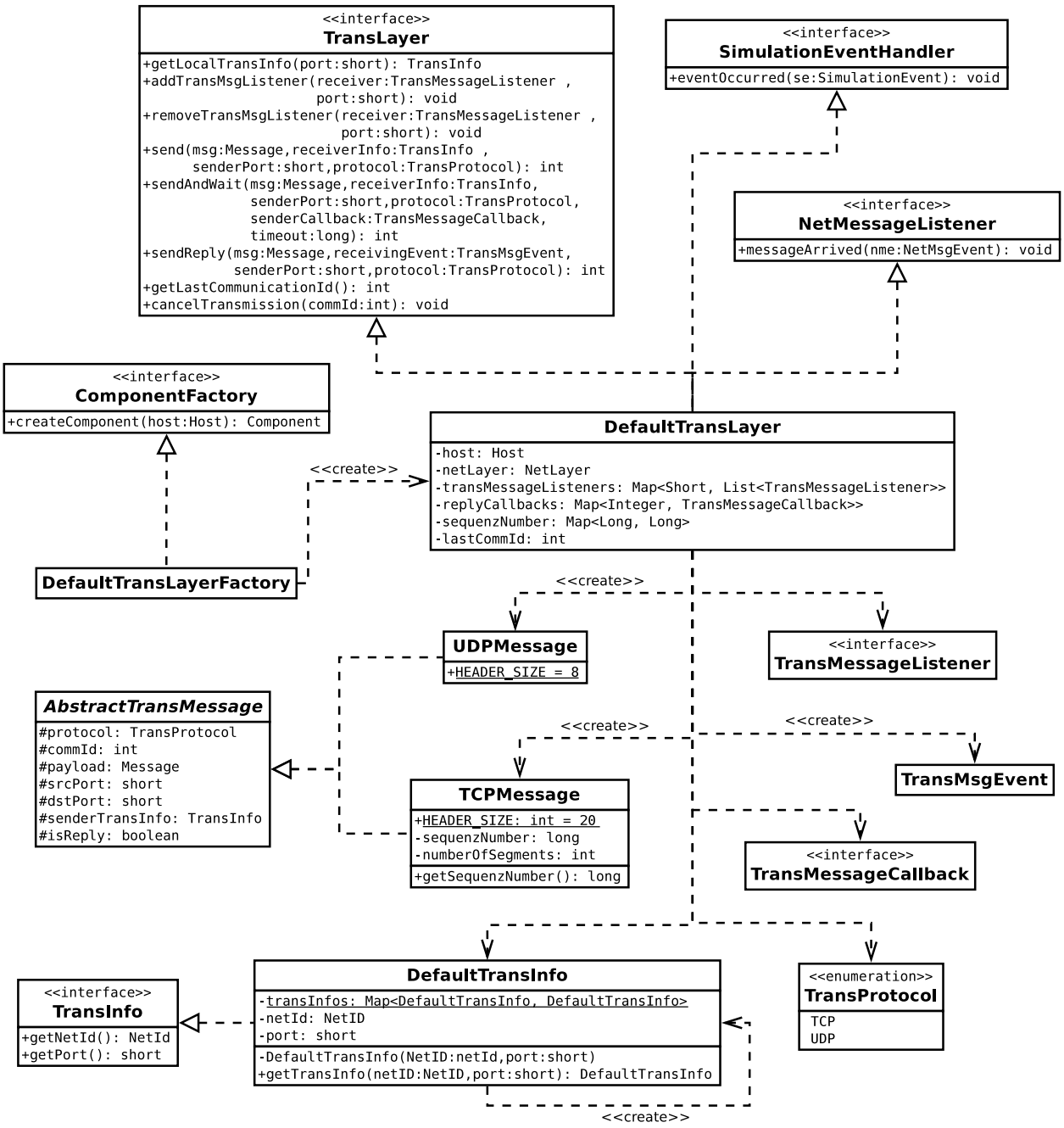


Figure 3.8: The classes of the default Transport Layer implementation and their dependencies.

[43] de.tud.kom.p2psim.impl.transport.DefaultTransInfo

`DefaultTransInfo`[43]: This is the default implementation of the interface `TransInfo`[44]. It is a container class that encapsulates a given instance of the class `NetID`[45] bundled with a port number for the aforementioned multiplexing purpose. For optimization reasons, `DefaultTransInfo` does not allow the creation of instances via a public constructor. Instead, the static method `getTransInfo(NetID,short)` is used to obtain an instance of this class. This way, it is assured that only one instance of a Host's `TransInfo` is created for the whole simulation. For large-scale P2P simulations this turned out to be important for the following (simple) reason. Typically, each peer holds a routing table including a large number of `TranInfo` instances to communicate with other peers. Since a peer may be known by a large number of other peers, meaning it's `TranInfo` is part of many routing tables, a large number of similar instances of the node's `TranInfo` would be created and thus a considerable amount of memory is wasted. The optimization helped to reduce the memory consumptions for large simulations significantly.

`TCPMessage`[46]: This is the default implementation of a TCP message. Beside the obligatory implementation of the interface `AbstractTransMessage`, the class provides one extra method to retrieve the sequence number of a TCP message and it holds the public constant `HEADER_SIZE` set to 20 Bytes. The protocol type of this message is assigned to `TransProtocol.TCP`.

`UDPMessage`[47]: This is the default implementation of a UDP message. It implements the interface `AbstractTransMessage` and holds the public constant `HEADER_SIZE` set to 8 Bytes.

`DefaultTransLayer`[48]: This is the core class of the Default Transport Layer's implementation. It implements the interface `TransLayer` and provides basic Transport Layer functionality to higher layers.

For de-multiplexing incoming messages to registered instances of `TransMessageListener`, the class implements the interface `NetMessageListener`[49] and registers itself as listener at the Network Layer. It accepts incoming Network Layer messages, strips of their Network Layer headers, and passes the messages to all listeners registered for the ports they were sent to. As mentioned before, this mechanism is an implementation of the Observer Design Pattern [10] and used at all layers of PeerfactSim.KOM. An overview about the communication inside a Host, which includes this mechanism, can be found in Subsection 2.2.2.

The default Transport Layer class also realizes the `sendAndWait` concept described in Subsection 3.2.2. For this purpose it maintains a collection of `TransMessageCallback` instances to dispatch incoming reply messages. To realize the timeout for unanswered requests, `DefaultTransLayer` implements the interface `SimulationEventHandler`[50] and schedules timeout simulation events, providing itself as callback.

For message transmission, the Default Transport Layer allows to choose between the protocols UDP and TCP. They are realized using the aforementioned message classes `UDPMessage` and `TCPMessage`. The transmission of UDP messages is very simple and does not include any further logic beside passing the message to the Network Layer and dispatching it to the right `TransMessageListener` on receiving. For the transmission of TCP messages, the Transport Layer maintains distinct sequence numbers for each connection. These are transferred as part of the TCP message and are used to assure a correct ordering when passing the messages to higher layers. The implementation of this mechanism is not very sophisticated and relies on the assumption that a TCP message may be delayed by the subnet, but is delivered at some point in time. There is no mechanism within this Transport Layer implementation to deal with a loss of TCP messages. This would require more complex mechanisms and the initiation of retransmissions. At the moment, lost TCP messages would result in a potentially growing queue of messages, which

[44] `de.tud.kom.p2psim.api.transport.TransInfo`
[45] `de.tud.kom.p2psim.api.network.NetID`
[46] `de.tud.kom.p2psim.impl.transport.TCPMessage`
[47] `de.tud.kom.p2psim.impl.transport.UDPMessage`
[48] `de.tud.kom.p2psim.impl.transport.DefaultTransLayer`
[49] `de.tud.kom.p2psim.api.network.NetMessageListener`
[50] `de.tud.kom.p2psim.api.simengine.SimulationEventHandler`

would be never delivered to any `TransMessageListener`.

`DefaultTransLayerFactory`[51]**:** This class implements the interface `ComponentFactory`[52] and is responsible to instantiate this Transport Layer's main class (`DefaultTransLayer`). The factory is necessary to allow the configuration of simulation scenarios using this Transport Layer implementation.

3.2.3.2 Configuration

The configuration of the implemented Transport Layer is straightforward, as it does not provide any configurable properties. The configuration using the XML configuration file includes two steps. The first is to define a name for the layer and referencing the factory class. Figure3.38 shows an example for such a definition, while during the second step, the Hosts are configured with the aforementioned Transport Layer (we refer to Chapter 5 for a detail description of the configuration process).

```
1 <Configuration>
2   [...]
3    <TransLayer class="de.tud.kom.p2psim.impl.transport.DefaultTransLayerFactory"/>
4   [...]
5 <Configuration>
```

Listing 3.38: Example for the definition of the Transport Layer

[51] `de.tud.kom.p2psim.impl.transport.DefaultTransLayerFactory`
[52] `de.tud.kom.p2psim.api.common.ComponentFactory`

3.3 Overlay Layer

This part of the chapter describes the Overlay Layer. Here you can find a description of several P2P-overlay implementations that come with the simulator. To be able to use an overlay implementation, the class that defines the behavior of a peer has to implement at least the interface `OverlayNode`[53] or the abstract class `AbstractOverlayNode`[54].

The following subsections contain information about the existing overlay implementations. Each of them firstly gives some general information about the specific overlay concepts, then deals with the implementation for the simulator, gives an overview about the possible configuration parameters and finally describes the API for using the overlay.

3.3.1 Gnutella-like overlays

Gnutella belongs to the class of *unstructured* overlays, regarding the storage and retrieval of resources as well as regarding the selection of neighbors. In the first version of Gnutella [2], a peer may arbitrarily connect to other peers. New peers are discovered through multi-hop ping discovery mechanisms. A query is simply flooded from one peer to another with a Time-to-Live (TTL) between 2 to 6. Since the basic Gnutella overlay scales badly, some improvements have been proposed and are used in practice. These derivatives of the Gnutella overlay are drawn together in the package `gnutella`, while the first version Gnutella 0.4 can be found in the package `gnutella04`. Both packages are situated in the `overlay`-package[55]. From the application perspective, the behavior is equal, so Gnutella 0.4 as well as its derivatives use the same API, described in the following subsection.

3.3.1.1 The Gnutella API

The Gnutella API[56] offers basic connectivity methods like `join()`, which connects a peer with the overlay and `leave()` which gracefully leaves the overlay (in contrast to spontaneous disconnection). Furthermore, it provides common procedures for publishing and querying documents, which are situated in the `api`-package[57].

Interface `IResource`

Classes that implement `IResource` may be published in a Gnutella overlay and are available for queries of foreign Hosts. The interface `IResource` has just one field, `getSize()`, which represents the transport size of the resource, while it is transmitted. A basic implementation of this class is `RankResource`, which just represents a resource by its rank, an Integer ID.

Interface `IQueryInfo`

`QueryInfo` encapsulates all information needed for matching of resources in the querying process of the Gnutella overlay. The method `getNumberOfMatchesIn(Set<IResource> resources)` looks in the given set of `IResource` implementations for matching resources and returns the number of them. A basic implementation is `SingleResourceQuery`, which takes an `IResource` object as an argument in its constructor, and returns 1 if the given set contains an equal object (according to `equals()`) and 0 otherwise.

[53] `de.tud.kom.p2psim.api.overlay.OverlayNode`
[54] `de.tud.kom.p2psim.impl.overlay.AbstractOverlayNode`
[55] `de.tud.kom.p2psim.impl`
[56] `de.tud.kom.p2psim.impl.overlay.gnutella.GnutellaAPI`
[57] `de.tud.kom.p2psim.impl.overlay.gnutella`

`publishSet(...)`

This method publishes the given set of `IResource` implementations in the overlay. For most of the implementations, it is more efficient to publish an entire set instead of publishing every resource for itself, since often resources are published in groups and potential update mechanisms can be triggered after the last element in the set has been published, instead of triggering it after every single element published.

`query(...)`

This method starts a new query originating from the owner of the method. It takes a `QueryInfo` object and the number of hits wanted for this query. If the wanted number of hits has been reached, the Gnutella implementation is allowed to stop the query process.

Event processing

The class `GnutellaEvents`[58] works as a dispatcher for Gnutella-specific overlay events, like queries succeeded etc. There are two possibilities of using it regarding the scope of the dispatcher: globally or locally. The global dispatcher triggers an event for every Gnutella peer in the overlay, while local dispatchers are bound to a particular node and only trigger the events occurring in this node. The global dispatcher can be accessed via `GnutellaEvents.getGlobal()`, while the local dispatcher of any Gnutella node can be accessed via `GnutellaAPI.getLocalEventDispatcher()`. Once you have the dispatcher you can add your own event listeners to it implementing the interface `IGnutellaEventListener`[59].

3.3.2 Gnutella 0.6

One of the improvements of the original Gnutella overlay (Gnutella 0.4) is Gnutella 0.6 [16], implemented by clients like LimeWire[60]. In the Internet, the amount of available bandwidth at the peers is very heterogeneous. In Gnutella 0.6, we therefore split the network into two types of nodes, the *ultrapeers* and the *leaves*. Peers with only a little bandwidth will become leaves, the rest that is capable of carrying the large amount of Gnutella overlay traffic will become ultrapeers. Leaves connect to ultrapeers like clients connect to a server. To support the robustness of the overlay regarding the connection of peers, leaves connect to multiple ultrapeers (e.g. 3). If an ultrapeer fails, the leaf is still connected to two other ultrapeers and does not lose the connection to the overlay. If a leaf connects to an ultrapeer, it tells the ultrapeer about the resources it has shared. The ultrapeer stores information about the leaf's resources in a *query routing table* (QRT)

When a leaf starts a query, it forwards the query to one or multiple ultrapeers it is connected to. If an ultrapeer receives a query from a leaf, it starts a query procedure called *Dynamic Querying* at its immediate ultrapeer neighbors. Dynamic Querying successively does the following steps:

- **Browsing of leaves**: The ultrapeer looks in the query routing table of its leaves if enough hits are found. If this is true, the query stops here.
- **Probe Query**: The ultrapeer starts a shallow search (a low TTL, e.g.1) that is flooded to all of its ultrapeer neighbors. If enough hits are received, the query stops here.
- **Controlled Broadcasting**: If the hits are still insufficient, the ultrapeer submits a deeper query to one of its neighbors. If the query did not bring sufficient results, a query is submitted to the next neighbor and so on. If this query was forwarded to all neighbors without sufficient results, the process finishes unsuccessfully.

[58] `de.tud.kom.p2psim.impl.overlay.gnutella.api.evaluation`
[59] `de.tud.kom.p2psim.impl.overlay.gnutella.api.evaluation.GnutellaEvents`
[60] http://wiki.limewire.org/index.php?title=How_Gnutella_Works

3.3.2.1 Implementation

The implementation of the Gnutella 0.6 overlay is oriented on the behavior of the LimeWire client. Instead of the HTTP-like ASCII communication of the Gnutella protocol, it uses UDP and binary representation of values transmitted, which are less traffic-consuming. The implementation uses *Pong Caching* to discover new nodes.

There are some workarounds to face common problems we discovered especially in the simulation of the Gnutella 0.6 behavior: In an environment without churn, ultrapeers tend to connect to each other until the maximum amount of connections is reached. Now, if another ultrapeer wants to join the network, there would not be any room for it to connect to. Therefore, we added another policy: If an ultrapeer that is badly connected (a low degree, e.g. < 5) wants to connect to an ultrapeer that does not have any free connection slots, the latter ultrapeer will drop an existing connection and will tell the peer that was dropped about the poorly-connected peer the connection was dropped for. The node that was dropped can now connect to the poorly-connected node, as well. So, new poorly-connected peers break up connections and are inserted *between* these connections.

Mind that this kind of implementations has some security flaws. A successful Denial-of-Service (DoS) attack would be to connect to victim peers multiple times along with advertising poor connectivity, which will then drop existing connections and end up connectionless. If you want to disable this feature, set the configuration variable `RANDOMLY_KICK_PEER` to `false`.

3.3.2.2 Usage

Gnutella 0.6 can be controlled by the common Gnutella API described in Subsection 3.3.1.1, thus there should be no differences between the offered functionality and methods of Gnutella-based overlays.

3.3.2.3 Configuration

The configuration of Gnutella 0.6 in the configuration file is very simple as displayed in Listing 3.39.

```
1 <Configuration>
2   [...]
3   <NodeFactory
        class="de.tud.kom.p2psim.impl.overlay.gnutella.gnutella06.Gnutella06Factory"
        />
4   [...]
5 <Configuration>
```

Listing 3.39: Example for the configuration of Gnutella 0.6

The simplicity of this configuration results from the fact that the different values for the parameter of the overlay are loaded from a separate configuration class. The default Gnutella configuration[61] is extended with the `IGnutella06Config`-interface[62]. The interface sets up Gnutella 0.6 with the following parameters:

- `CONNECT_TIMEOUT` - Timeout of a connection attempt to another peer - **Default value: 1s**
- `MAX_LEAF_TO_SP_CONNECTIONS` - Maximum connection count of a leaf to ultrapeers - **Default value: 3**
- `MAX_SP_TO_LEAF_CONNECTIONS` - Maximum connection count of an ultrapeer to leaves - **Default value: 30**

[61] `de.tud.kom.p2psim.impl.overlay.gnutella.common.IGnutellaConfig`
[62] `de.tud.kom.p2psim.impl.overlay.gnutella.gnutella06`

- MAX_SP_TO_SP_CONNECTIONS - Maximum connection count of an ultrapeer to other ultrapeers - **Default value: 32**
- PONG_CACHE_SIZE - Size of the pong cache of ultrapeers - **Default value: 5**
- STALE_CONN_ATTEMPTS - Number of subsequent unsuccessful communication attempts with a neighbor before it is dropped - **Default value: 1**
- TRY_PEERS_SIZE - Number of contacts that are delivered to poorly connected peers along with the connection response (TRY_ULTRAPEERS) - **Default value: 10**
- PING_INTERVAL - Time between two ping attempts - **Default value: 15s**
- PING_TIMEOUT - If a neighbor has not answered a ping with a pong within this time, the peer is marked as stale - **Default value: 1s**
- RANDOMLY_KICK_PEER - Specifies if a random peer may be kicked when a poorly-connected peer wants to connect - **Default value: true**
- RANDOMLY_KICK_PEER_PROB - The probability that a random peer is kicked to make room for poorly-connected ones - **Default value: 20%**
- TRY_PEERS_ADD_LIMIT - If a peer has more connections than this value, it will just add nodes discovered via pong caching and will not accept any peers received by the TRY_ULTRAPEERS field. - **Default value: 12**
- ULTRAPEER_RATIO - The ratio in percent(!) of ultrapeers between the MayBeUltrapeerBandwidth and MustBeUltrapeerBandwidth limit - **Default value: 90**
- BOOTSTRAP_INTVL - The bootstrap interval between two bootstrap attempts of the same node. - **Default value: 5s**
- RESPONSE_TIMEOUT - Response timeout for various message types - **Default value: 1s**
- QUERY_DEPTH - The value for TTL for a query, until it is dropped - **Default value: 2**
- CONTROLLED_BCAST_STEP_DURATION - The duration of the controlled broadcast step in the dynamic query operation. - **Default value: 1s**
- PROBE_QUERY_DURATION - The duration of the probe query phase - **Default value: 1s**
- REQUIRED_HIT_COUNT - The required hit count that is necessary to mark a query as succeeded - **Default value: 1**
- LEAF_QUERY_DURATION - The duration, a leaf waits for a response when it has requested an ultrapeer to do its query - **Default value: 40s**
- QUERY_CACHE_TIMEOUT - The time a query UID is hold in the query cache to avoid duplicate relays of the same query - **Default value: 30s**
- MAY_BE_ULTRAPEER_BANDWIDTH - The bandwidth below which a node has to become a leaf - **Default value: 100KByte/s upstream and downstream**
- MUST_BE_ULTRAPEER_BANDWIDTH - The bandwidth above which a node has to become an ultrapeer - **Default value: 2MByte/s upstream and downstream**
- CONSIDER_ONLY_LAST_ENTRY - If only the last entry of a received pong cache shall be taken into account as a newly discovered peer. - **Default value: false**

3.3.3 Gia

Gia [6] enhances Gnutella-like overlays by allowing the peers to adapt their duties for the overlay depending on their capabilities (in terms of e.g. available bandwidth). Unlike Gnutella 0.6 (see Subsection 3.3.2), there is no classification into ultrapeers and leaves, instead, we assign a capacity value to every peer in the overlay. The capacity value is used in the connection establishment procedure and in the query relaying step.

Gia's protocol is more complex than Gnutella 0.6, but allows for a smoother load balancing between the bandwidths of the peers. The key concepts of Gia can be summarized by the following aspects:

- **Topology Adaptation**: This mechanism tries to connect low-capacity, poorly-connected peers with other peers that have a high capacity and are well-connected to many neighbors. This ensures that low-capacity peers can reach well-connected ones in only a few hops.
- **One-hop Replication**: This mechanism replicates the documents stored at every peer to its immediate neighbors. Poorly-connected peers may still have a lot of useful documents, but do not have the bandwidth to answer every query. Since Topology Adaptation ensures a good connectivity of low-capacity peers to ones capable of relaying many queries, the strong peers can answer queries with the replicated content of low-capacity peers, thus saving a lot of expensive bandwidth needed to forward queries to them.
- **Biased Random Walk** relays an incoming query only to one neighbor instead of every one of them, like it is done in other Gnutella overlays [2, 16]. Flooding techniques waste bandwidth at every peer to achieve network coverage. The random walk is "biased", because it tends to use the neighbor with the greatest capacity available as the next hop. Since this neighbor obviously knows more peers than low-capacity ones (thus more replicated content), a query can reach a greater coverage than taking a random peer as the next hop.
- **Flow Control**: Since all queries are directed to high-capacity peers, they may tend to get overloaded. This problem is avoided by assigning flow-control tokens to neighbors. If the tokens of a neighbor are depleted, no query may be forwared to this neighbor any longer. Additionally, queries arriving too fast are queued and scheduled, while queries from high-capacity neighbors are preferred by the scheduling mechanism.

3.3.3.1 Implementation

Regarding the mechanisms used for peer discovery, Chawathe et al.[6] do not provide any details. Therefore, our implementation uses *pong caching* to discover new nodes and fills a *Host cache* with the new nodes. To connect peer X with a peer Y, Gia uses a three-way handshake protocol. During the protocol, X and Y exchange sufficient information about each other to decide if booth accept the connection.

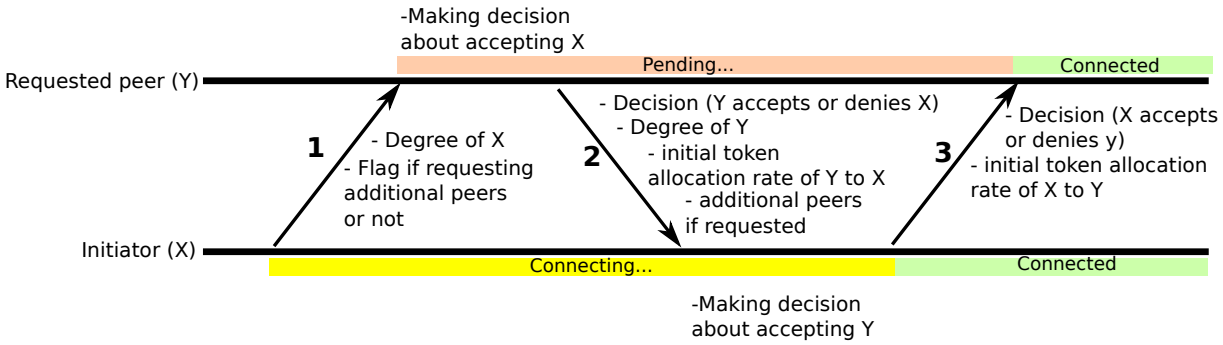


Figure 3.9: Three-way handshake of Gia for connecting two peers with each other

Figure 3.9 illustrates the protocol interaction between peer X and peer Y. Peer X has discovered Y and initiates the handshake. X sends (1) its current degree (number of neighbors currently connected) and a flag whether it wants to receive additional contacts from Y or not (if X has low connectivity, this flag is usually set). Y then makes a decision about accepting Y based on "Algorithm 1" [6]. Y then tells X (2) about the decision it has made about accepting X, along with Y's degree. The third value Y sends in handshake message (2) is the initial token allocation rate. This value only needs to be sent if Y has accepted the connection. If X has requested additional contacts, Y will deliver some of them as the fourth value sent to X in order to give X the chance to fill its Host cache. If the connection is not accepted, the

protocol stops here. Else, X makes a decision about accepting Y like Y itself did for X in the last step. X then sends a last acknowledgement (3) to Y, which contains its decision and the initial token allocation rate for Y, if accepted.

3.3.3.2 Usage

The Gia overlay can be controlled by the common Gnutella API described in Subsection 3.3.1.1, thus there should be no differences between Gnutella-based overlays regarding the handling.

3.3.3.3 Configuration

The configuration of Gnutella 0.6 in the configuration file is very simple as displayed in Listing 3.40.

```
1 <Configuration>
2   [...]
3    <NodeFactory class="de.tud.kom.p2psim.impl.overlay.gnutella.gia.GiaFactory" />
4   [...]
5 <Configuration>
```

Listing 3.40: Example for the configuration of Gia

The simplicity of this configuration results from the fact that the different values for the parameter of the overlay are loaded from a separate configuration class. The default Gnutella configuration[63] is extended with the interface `IGiaConfig`[64]. The interface sets up Gia with the following parameters (as far as possible, the default values are set to the ones supplied in [6]:

- `CONNECT_TIMEOUT` - The timeout for connection attempts - **Default value: 1s**
- `PONG_CACHE_SIZE` - The size of the pong cache - **Default value: 5**
- `STALE_CONN_ATTEMPTS` - The number of unsuccessful connection attempts before a contact is marked as stale and dropped - **Default value: 1**
- `PING_INTERVAL` - Time between two ping attempts - **Default value: 15s**
- `PING_TIMEOUT` - Timeout of a ping request - **Default value: 1s**
- `BOOTSTRAP_INTVL` - The bootstrap interval between two bootstrap attempts of the same node - **Default value: 5s**
- `RESPONSE_TIMEOUT` - The maximum response time allowed for various requests - **Default value: 1s**
- `REQUIRED_HIT_COUNT` - The required hit count that is necessary to mark a query as succeeded - **Default value: 1s**
- `QUERY_CACHE_TIMEOUT` – **Default value: 30s**
- `CONNECT_DECISION_HYSTERESIS` - When Gia looks for another peer that can be dropped for the one that attempts to connect, it compares the degree of these two peers. If the degree of the candidate to drop is larger than the degree of the attempting peer plus `CONNECT_DECISION_HYSTERESIS`, the candidate is dropped. Avoids flipping back and forth between two nodes with a similar degree - **Default value: 5**
- `HANDSHAKE_TIMEOUT` – **Default value: 1s**
- `MAX_NBRS` - Maximum number of neighbors a peer may have - **Default value: 128**
- `MIN_ALLOC` - The min_alloc parameter [6]. The paper says: "Finest level of granularity into which we are willing to split a node's capacity." - **Default value: 4**
- `MIN_NBRS` - Minimum number of neighbors a peer should have. - **Default value: 3**
- `ADAPTATION_MAX_INTERVAL` - The maximum interval between two adaptation attempts. If a peer is fully satisfied with its neighborhood, it uses this adaptation interval - **Default value: 10s**

[63] `de.tud.kom.p2psim.impl.overlay.gnutella.common.IGnutellaConfig`
[64] `de.tud.kom.p2psim.impl.overlay.gnutella.gia`

- ADAPTATION_AGGRESSIVENESS - The daptation aggressiveness defines how fast the adaptation interval shrinks with lower satisfaction: $adaptationInterval = adaptationMaxInterval * adaptationAggressiveness^{-(1-satisfactionLevel)}$ - **Default value: 64**
- HOST_CACHE_TIMEOUT - After this time in the Host cache, a contact is dropped. - **Default value: 50s**
- TRY_PEERS_ADD_LIMIT – **Default value: 5**
- TRY_PEERS_SIZE – **Default value: 10**
- REPLICATION_DELAY - The delay of the One-hop replication being initiated after a connection has succeeded - **Default value: 50ms**
- QUERY_BANDWIDTH_LIMIT_QUOTA - The part of the bandwidth of the node that may be used solely for queries - **Default value: 0.2**
- TOKEN_ALLOCATION_BANDWIDTH_QUOTA - The part of the bandwidth of the node that may be used solely for queries, as it is assumed by the token allocation. Should be slightly lower than QUERY_BANDWIDTH_LIMIT_QUOTA - **Default value: 0.1**
- TOKEN_BUCKET_SIZE - The size of the token buckets that regulate the flow of queries from every neighbor - **Default value: 5**
- MAX_QUERY_QUEUE_SIZE - Returns the maximum query queue size. If a query queue gets larger than this value, packets will be dropped - **Default value: 10**
- QUERY_TTL - The time to live of random-walk queries - **Default value: 512**
- MAX_TIME_IN_QUERY_QUEUE - Returns the maximum time a packet may stay in the query queue. A packet that stays in the queue for a longer time is dropped - **Default value: 10s**
- CONTACT_ACTIVITY_TIMEOUT - In Gia, only active neighbors share the bandwidth of a peer for the query relaying. A peer becomes active if it is sending queries, and becomes inactive again after the period given here - **Default value: 15s**
- TOKEN_THROTTLE_QUOTA - If the query queue of a neighbor gets longer than ThrottleTokenQueue-Size, its token allocation rate is lowered by the given value. - **Default value: 0.2**
- THROTTLE_TOKEN_QUEUE_SIZE - If the query queue of a neighbor gets longer than the given queue size, its token allocation rate is lowered by TokenThrottle Quota. - **Default value: 7**
- QUERY_TIMEOUT - If a query gets older than this value, the initiator will give up waiting for QueryHits. - **Default value: 20s**
- HOST_CACHE_SIZE - Returns the size of the Host cache. - **Default value: 15**
- DOWN_BW_PER_CAP_POINT - For every DownBWPerCapPoint a peer has (in bytes/sec), the peer gets one capacity point. Then the minimum of upstream bandwidth and downstream bandwidth capacity is taken - **Default value: 2000 bytes/sec**
- MIN_CAPACITY - Capacity Assignment: Returns the minimum capacity that may be assigned to a peer - **Default value: 1**
- UP_BW_PER_CAP_POINT - For every UpBWPerCapPoint of its upstream bandwidth (in bytes/sec), the peer gets one capacity point. Then the minimum of upstream bandwidth and downstream bandwidth capacity is taken - **Default value: 2000 bytes/sec**

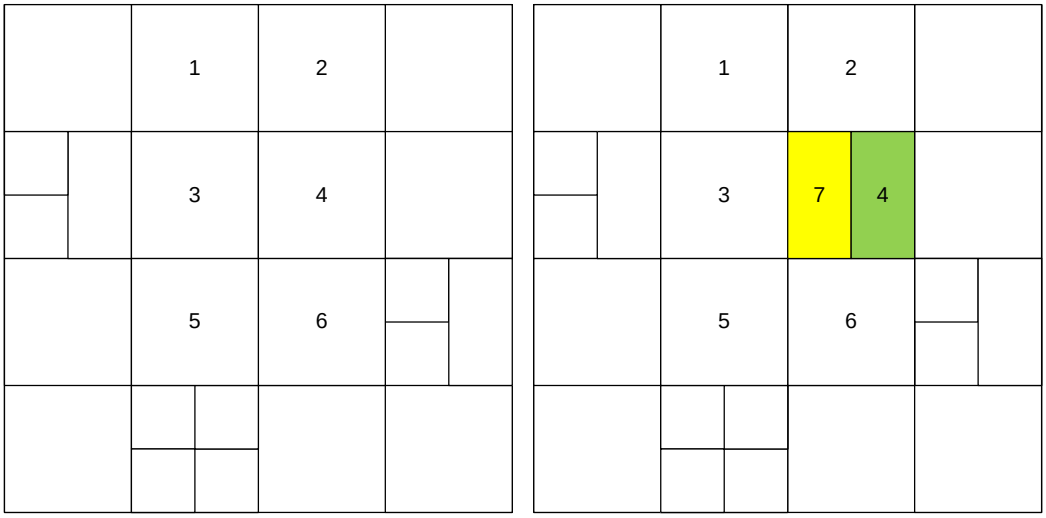
3.3.4 CAN

The following section contains a description of the central idea and the implementation of the Content-Addressable Network (CAN). It is based on the publication by Ratnasamy et al. [28].

This part explains the basic concepts of the CAN. The first subsection 3.3.4.1 discusses the implementation within the simulator. The two latter sections 3.3.4.2 and 3.3.4.3 are hints how to configure and use the overlay within the simulator.

The CAN is a member of the distributed-hash tables (DHT) family. The idea of DHTs is to save the keys and values of the hash tables distributed over all nodes. Therefore, each node has an own zone, which saves a part of the hash table. In [28], this indexing system of the DHT is called CAN.

The CAN distributes the nodes in a d-dimensional Cartesian coordinate space. The virtual coordinate space is used to store the key-value pairs. Every pair has its own fixed position. This position is determined from the key. In the following, the SHA-1 [4] hash function is used to resolve the key from the data name, but any kind of hash function which gives a unique key can be used. Figure 3.10a shows an example of a CAN. A 2-dimensional coordinate space is portioned by 23 nodes. Each node has its own zone and is responsible for all keys within this zone. The hash value of the network address is used to give the nodes a unique ID for the overlay, therefore the SHA-1 is used as well.



(a) A CAN with 23 nodes **(b)** Node 7 joined the overlay

Figure 3.10: Restructuring of a CAN overlay

Topology

As already mentioned, the topology of the overlay is a d-dimensional Cartesian coordinate space. In this space every node gets its own zone. To reduce the size of the routing table, which is stored in each node and to make the overlay easily scalable, every node just knows its direct neighbors. In a d-dimensional coordinate space two nodes are neighbors if their zones overlap in d-1 dimensions and abut along one dimension. If the nodes are uniformly distributed over the space, the amount of neighbors is almost equal for every node. The nodes know required information of their neighbors, including overlay ID, IP-address and owned zone. Due to the small size of the routing table, only a small amount of nodes are affected if something changes in the network. The CAN overlay is self-organizing, no central node is needed to organize the overlay. The nodes learn about their neighbors and organize them during the join process.

Join

A new joining node must be allocated to its own zone. This is done by finding another node and splitting its zone. The joining process takes three steps:

1. Find a node in the overlay.
2. Send a message towards this node.
3. After receiving a join message a node splits its zone and the new joining node gets the half of it.

In order to find a node in the network, a so called bootstrap is used. Such a bootstrap saves the information of all or a part of all nodes in the overlay. In [28] one suggested bootstrap mechanism is YOID [9]. The joining node looks up a node from the overlay by contacting the bootstrap, which returns the information of a randomly chosen node. Note that if every node is chosen randomly, the possibility

is high that the zones within the CAN are not uniformly distributed. Therefore, one improvement could be to take the node with the biggest zone.

The joining node sends a join message to the picked node. The picked node divides its own zone in two parts. It takes one half itself and hands the new part over to the joining node. The zones are split by special rules depending on the number of dimensions. For example in a two dimensional space, the zone is first split along the x-coordinate and then by the y-coordinate. Another opportunity to distribute the nodes uniformly is introduced in [28]. Here, the node, which receives the join message compares its zone with its neighbor zones. If one of these zones is bigger then its own, the join message will be forwarded to this zone.

During the joining process, not only the zones are split, but the neighbors and the keys are shared between the old and the new node as well. After this process every node should have the right set of neighbors and hash keys. Moreover, the neighbors have to be informed that, due to the joining process, the old node changed and a new arrived.

An example for the joining process is given in Figure 3.10b. This figure shows the same CAN as Figure 3.10a does, but node 7 joined the overlay and node 4 divided its zone to hand half of it over to node 7.

Leaving of a Host and takeover

If a node wants to leave, it simply hands its zone and the included hash keys over to one of its neighbors. If the zone of the neighbor can be merged with the handed zone, it is done. Otherwise the neighbor temporarily handles the second zone and starts the zone-reassignment which is described in the following paragraph.

Every node must ensure that all neighbors are reachable, therefore, they periodically send messages to all neighbors. If one node ungracefully leaves the overlay, it is taken over by one of its neighbors. Like above the node tries to merge the zone with its own or temporarily handles the zone and starts the zone-reassignment.

Zone-reassignment

If one node has to handle two zones, it starts the zone-reassignment procedure. That means it reassigns the zones such that every node has just one zone. Besides the representation in the d-dimensional Cartesian coordinate space, the structure of the CAN can be seen as a *partition tree*. The children of every vertex in the tree are the zones which were split, so they are siblings. That means they have *common parents*, and therefore, they share the path in the tree from the root to the vertex of the siblings. In addition to the overlay ID, every node has a virtual identifier (*VID*) to set the place of its zone in the *partition tree*. If a node splits up its zone, it gives its children the own VID and appends either *0* or *1* to the end. In the 2-dimensional Cartesian coordinate space, the zones are firstly parted along the x-dimension and then along the y-dimension. If a zone is parted along the x-dimension, the node in the left zone gets the *0* and the other node the *1*. Along the y-dimension, the node of the upper zone gets the *0* and the lower the *1*. Each node also knows its VID neighbors in the tree. So they can route through the *partition tree*.

An example of a *partition tree* is shown in Figure 3.11. In the left part of the figure the 2-dimensional coordinate space is depicted, while the right part displays the *partition tree* of the zones. For example take the zones of node 10 and 11, they have both the same parents, so all numbers except the last in the VID are the same (the common VID values are *111*), that means they originated from the same zone.

If a node has two zones and it wants to reassign them, there are two different ways:

- In the first way the sibling of the zone is as well a leaf. Then the merging of the zones is easy, the two zones are just merged to one and is handed to the former sibling. For example node 10 leaves, then node 11 will merge the zone with its own.
- The second way is a bit more complicated. If the sibling of the zone is not a leaf, a search for all leaves in this subtree has to be performed. For instance node 9 leaves, then a search through the siblings subtree has to be performed and it gives the leaves node 10 and 11. So node 10 and 11

merge their zones together with the zone of node 9 and both of them take the half of the new created zone.

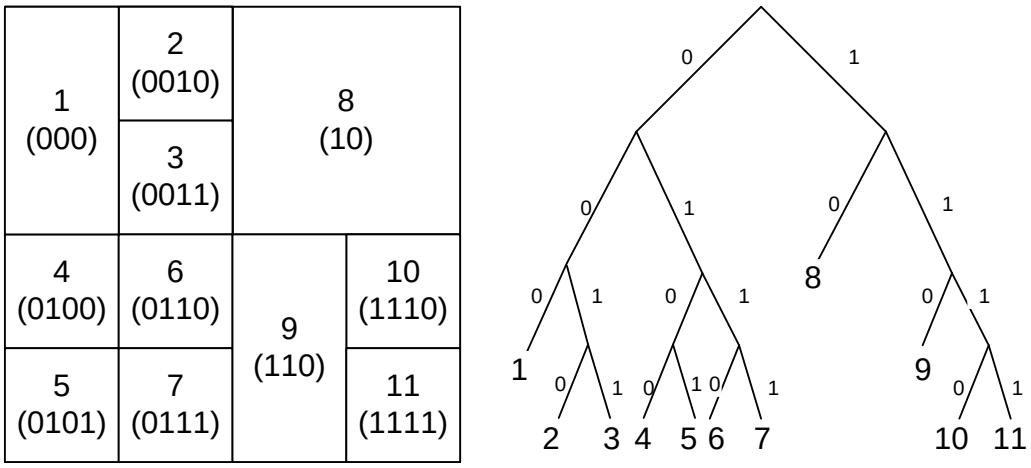


Figure 3.11: Tree structure for the zone-reassignment process.

Routing

Every node just knows its direct neighbors, therefore, there is no need for a strong routing algorithm to send messages. Every message includes the destination coordinates and it is just sent or forwarded to the neighbor, which is closest to the destination.

In [28] it is written that the average of the routing path should be $(d/4)(n^{1/d})$, where d is the dimension and n the number of total zones. The average number of neighbors is $2d$. That means, if the nodes a uniformly distributed over the coordinate space, the number of stored information in the nodes stays the same. Only the path length increases as $O(n^{1/d})$.

An advantage of the routing of CAN is that more than one path could be used. So if one path is broken or occupied another can be used. Even if one neighbor is not available another route to the destination could be used.

Store and lookup

If a node wants to store an object, it takes the object's hash key and sends a message through the overlay described by the routing mechanism above. The message includes the hash key and the information of the node, which wants to store the value. When the message arrives at the destination, the node, which is responsible for the destination zone, stores the hash key as the key and the contact information as the value. To be sure that no key is lost due to an unreachable node, the nodes have to refresh the key after a certain interval.

If a node wants to lookup a value, it works similar to the store process. The only difference is that, instead of saving, the destination node response with a message that contains the value to the corresponding key.

3.3.4.1 Implementation within the simulator

You can find the implementation of the CAN overlay in the package `can`[65]. In this section we do not describe everything of the implementation in detail, instead we focus on the main mechanisms and

[65] `de.tud.kom.p2psim.impl.overlay.dht`

the differences between [28] and this implementation. The implementing classes of the contemplated operations can be found in the `operations`-package[66].

Joining

The joining process is implemented as it is described above. The join operation is started in the `JoinOperation` class. The problem is that the nodes are chosen randomly, so they are not uniformly distributed over the whole space. It can be shown that a few nodes have quiet big zones and many nodes have just very small zones compared to the big ones. In this implementation two possibilities are given, either the nodes are randomly chosen or always the node with the biggest area is chosen. The latter approach distributes the nodes more randomly.

Leaving of a Host

The description above outlines that a leaving node hands over its own zone to one of its neighbors, which initializes the zone-reassignment. In the `LeaveOperation` class this is implemented in a bit different way. So the node does not hand over the zone, instead, it directly starts the zone-reassignment and distributes all newly created zones to the owners by its own.

Takeover

The takeover operation is implemented in a different way as it is described in [28], because this operation is not well specified. There are three different operations:

1. `TakeoverOperation` is started after a certain time interval and starts for every neighbor a `TakeoverReplyOperation`.
2. `TakeoverReplyOperation` sends a ping message to a certain neighbor to ensure that it is still reachable. If a node gets a ping message, it answers with a pong message, which includes all contact information, neighbors and VID neighbors. The operation is finished after the pong arrives. If the pong does not arrive, the node sends the ping a number of times again. If still no pong arrives, it assumes that the node is not available any longer. Now, the node, which started the `TakeoverReplyOperation` tries to find the closest VID neighbor based on the data, which is stored from earlier pong messages. That means that the node, which has the most common parents is searched. Either, it is already the node, which did not get the pong message, so it starts directly the `TakeoverRebuildOperation` or it sends a message to the closest neighbor to start the `TakeoverRebuildOperation`.
3. `TakeoverRebuildOperation` is responsible for the reassignment of the zones of the nodes. It is only possible to reassign the zone of the unreachable node, if enough information about this node is available. The node starts to search in the *partition tree* for every node that stores the most common parents to the unreachable node. After all of them where found the reassignment starts, as described before.

The problem of the complete mechanism is that all neighbors must have all the information about the unreachable node to reassign the zone of the unreachable node. This is not always possible, as for example, it is not possible to reassign the zones of two unreachable nodes, which are located next to each other.

Store and lookup

Store and lookup are implemented as described in Subsection 3.3.4. The store operation is started in the `StoreOperation` and the lookup operation in the `LookupOperation` class. Both of them send a message including the key and the own contact information into the overlay. For a store message, the destination node saves the hash values as the key and the contact information as the value. For a lookup message it either sends the value to a certain key back or it sends null if the key is not stored.

[66] `de.tud.kom.p2psim.impl.overlay.dht.can`

3.3.4.2 Configuring CAN within the simulator

To configure scenarios containing CAN as overlay, the factory class is `CanNodeFactory`[67]. This class does not provide any kind of configuration properties. To change any configuration of the overlay, this is done in the `CanConfig`[68] class. In the following the configuration parameters are described:

- `CanSize` - Defines the size of the CAN. The CAN is implemented as a 2-dimensional coordinate space, so the whole space will have a size of `CanSize`×`CanSize`.
- `overloadNumber` - This prepares the overlay for the improvement to overload the zones, which is described in [28]. That means every zone includes up to this value of nodes. The zone is managed by one central node. This function is just prepared, but not well tested.
- `distribution` - At the joining process the zones, which should be portioned are either chosen uniformly (`0`), randomly (`1`) or uniformly (`2`), as defined in [28]. Uniform means that always the biggest zone is chosen, while random returns one random zone. [28] introduces an opportunity to distribute the zones more uniformly. Therefore, every node that gets a join message compares its own zone with the zones of the neighbors. If a neighbor has a bigger zone, the join message is forwarded to this node.
- `VisSize` - Sets the window size of the visualization. Remember the visualization is just usable for small number of nodes.
- `lookupMaxHop` - The messages should not travel for an infinite number of hops through the overlay, because if they are sent to a dead zone, they will not reach their destination. This value gives the maximum hop number. This value is also valid for the store message.
- `waitTimeAfterLeave` - When a node wants to leave, it sends a leave message and has to wait until all the replies have arrived. After this time, the node starts the reorganization of the zones. This value can be a problem in nonuniform overlays, because the time can vary a lot until all reply messages arrived.
- `waitTimeBetweenPing` - Time between two ping messages. Remember that the ping could be send `numberPings` times.
- `timeout` - Sets the timeout of a ping message. After this time, either the ping is sent again, or the neighbor is declared as missing.
- `waitForTakeover` - Defines the time between a missed pong message and the start of the `TakeoverRebuildOperation`. All nodes with the same parents in the *partition tree* should send their information within this time interval.
- `waitTimeToStore` - Describes the interval in which the lookup or store messages should reach their destination and the sender receives the an acknowledgment. After that time the operation will timeout.
- `waitTimeToRefreshHash` - Due to unreachable nodes and lost hash keys, the hash key has to be refreshed after a certain time interval.
- `numberLookups` - If a lookup has not succeeded the peer will try it `numberLookups` times again.
- `numberPings` - If no pong message is received the node will send the ping `numberPings` times again.
- `statiticsOutputPath` - Defines the path of the output files for the evaluation files.
- `intervallBetweenStatistics` - Defines the time interval after that the data is written in the evaluation file.
- `waitTimeBetweenLookups` - Describes the time period between two consecutive lookups. It is implemented in `LookupPeriodically`[69] and is mainly used for the evaluation.

[67] `de.tud.kom.p2psim.impl.overlay.dht.can`
[68] `de.tud.kom.p2psim.impl.overlay.dht.can`
[69] `de.tud.kom.p2psim.impl.overlay.dht.can.operations`

3.3.4.3 The API of CAN for the simulator

The methods to start an operation for a node in the CAN overlay are defined in `CanNode`[70]. These methods can be called from the action file:

- `join(OperationCallback callback)` - This method starts the join operation for a node. The `OperationCallback`[71] handler is passed to the operation, but it is not needed, so a dummy could be used.
- `leave(OperationCallback callback)` - Instructs the node to leave the overlay. As before the `OperationCallback` is just a dummy.
- `giveNodeInfo(OperationCallback callback)` - This method is used for debugging of the overlay, it shows all information about the actual node in the console.
- `store(String obj, OperationCallback callback)` - Stores the hash of the given`obj` in the CAN. So it starts the `StoreOperation`. As before the `OperationCallback` is just a dummy.
- `valueLookup(String data, OperationCallback<DHTObject>callback)` - Looks the hash value of the committed String `data` up. Therefore it start the `LookupOpeartion`. As before the `OperationCallback` is just a dummy.
- `LookupPeriodically(OperationCallback<DHTObject> callback)` - This method is used for the evaluation of the overlay. It starts a `LookupOperation` every `waitTimeBetweenLookups` intervals. A random key is searched. As before the `OperationCallback` is just a dummy.

3.3.5 Chord

This section describes the overlay network Chord and its functionality [29]. The section is structured as follows. The first Subsection 3.3.5.1 briefly introduces the concept of Chord and Subsection 3.3.5.2 describes its implementation within the simulator. Afterwards, Subsection 3.3.5.3 details the configuration and Subsection 3.3.5.4 outlines the API of Chord for the action file.

3.3.5.1 Introduction

A fundamental problem in peer-to-peer (P2P) systems is to efficiently store and look up a particular data items. Chord belongs to the Distributed Hash Table (DHT) overlays and therefore provides support for store/lookup operations as follows: Each node has an unique overlay identifier (overlay-ID), represented as a 160-bit number. The location of data can be implemented by assigning each data item to a key, and storing the key/data item pair at the node, whose corresponding responsibility interval contains the key. To lookup the data item, the node queries the correspoding key, which results in a query that is routed towards the peer, whose responsibility interval contains the key.

Topology and routing table

Chord uses a one dimensional ID space. The IDs are arranged with increasing IDs on the ring. The ring can have IDs ranging from 0 to $2^m - 1$ (in our implementation $m = 160$). Each node has a successor and a predecessor. The successor to a node is the next node in the circle in the clockwise direction. The predecessor of a node is the next node in the circle but in the counter-clockwise direction. A node is responsible for a range between its own ID and the ID of its predecessor on the ring. Besides its successor and predecessor, each Chord node knows some short-cuts, so called fingers, to other nodes further away. The finger table has at most m entries (indexed from 0 to $m - 1$). The i^{th} entry in the table at node n contains the node ID of the successor of the key $n + 2^i$.

[70] `de.tud.kom.p2psim.impl.overlay.dht.can`
[71] `de.tud.kom.p2psim.api.common`

$$i^{th} finger = successor((nodeID + 2^i) \bmod 2^m)$$

The Figure 3.12 illustrates a Chord overlay with 10 peers. The ID is a 6-bit number ranging from 0 to 63. Each node has a finger table with 6 entries.

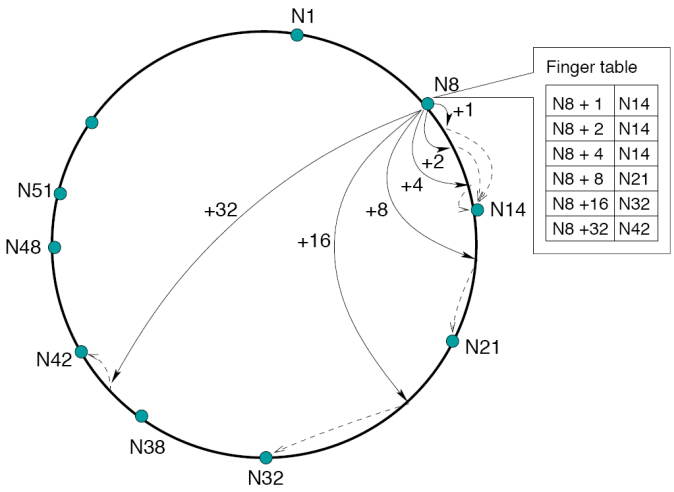


Figure 3.12: The Chord topology and its finger table [29]

Routing and Lookup

Lookups are routed clockwise through the network using the finger entries. The lookup is always sent to the closest finger, which is still preceding the queried ID. This step will be repeatedly performed until the query reaches the node which directly precedes the ID on the Chord ring. This node forwards the request to its successor s. Node s is the direct successor of the ID and is therefore responsible for the key. A simple lookup sample can be found in Figure 3.13. Node 8 forwards the query to node 42, since it is the closest peer not exceeding the queried ID. Then node 42 forwards the query to node 51. Node 51 knows that its successor node 56 is also the successor of the key and therefore is responsible for it. As a result, 3 steps are needed to find the successor for the key.

Stabilization

One of the most important tasks of structured P2P protocols is to maintain and keep the overlay structure stable. Stability is denoted by the correctness of the direct neighbors and the finger entries as well as by handling fast topology changes, due to joining and leaving nodes. By the simple stabilization mechanism, node n periodically sends stabilization messages to its successor s. Node s replies to this message by sending its predecessor entry p to node n. In the normal case, p should be equal to n. In the following, we will detail the cases if p does not equal n:

- p between n and s: It means that a new node p joined the network and s is no longer the successor of n. Node n sets p as its new successor and informs p that it is its predecessor.
- n between p and s: In this case, the node n just joined the network and s does not yet know about the existence of n. Therefore node n will contact node s and inform s that it is now its new predecessor. The next time, node p will update its successor as described in the previous case.

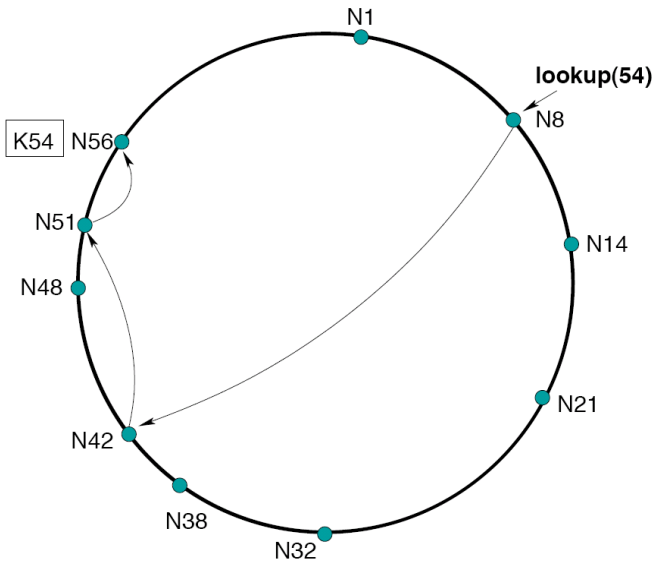


Figure 3.13: Routing and lookup mechanism in Chord [29]

3.3.5.2 Implementation within the Simulator

The implementation of Chord can be found in the package `chord2`[72]. This section gives an overview of the implementation and focuses on our improvements in comparison with the basic concept introduced in [29].

Joining of peers

The joining of a new node can be realized by calling the method `join()` in the class `ChordNode`[73]. The join operation contains the following steps:

- The peer n contacts an arbitrary peer n_0 in the network. Peer n_0 starts a query to find the successor s for peer n
- When n_0 receives a reply from successor s, n_0 forwards the contact info of s to n
- Peer n will directly send a handshake message to s, which replies to this message by sending the address of its predecessor p to node n
- Node n proves that its ID must be between the IDs of s and p. If not, n knows that node p just joined between it and its "out-of-date" successor s. For this reason, node n sends a new handshake message to p and waits for its reply
- In a normal case, the ID of n is between the IDs of s and p. Peer n sets s as its initial successor and p as initial predecessor. At the same time, n will inform both s and p of its participating. The join process is finished

Additionally, peer s sends to the new node n a list of peers retrieved from its finger table. The list is useful for peer n to initialize its finger table. As a result, the costs for initialising the finger table will be decreased and the stability enhanced.

Stabilization

Every node runs the stabilize phase periodically, which is described in the previous subsection. The method `execute()` in the `UpdateDirectSuccessorOperation`[74] class is used to perform a stabilization

[72] `de.tud.kom.p2psim.impl.overlay.dht`
[73] `de.tud.kom.p2psim.impl.overlay.dht.chord2`
[74] `de.tud.kom.p2psim.impl.overlay.dht.chord2.callbacks`

operation. There are several aspects, which should be paid more attention, if node p runs a stabilize operation and receives no reply from its successor s. It may try to send stabilization messages several times, but after a certain amount of times, node n should suppose that s is no longer online. The ring will be broken if n does not know about no other successors. Therefore the list of successors and predecessor is introduced. If the direct successor fails, it will be dropped from the list and the next successor replaces the older one.

Obviously, the list of successors and predecessors has to be regularly updated as well. By updating the i^{th} successor s_i, node n sends a `RetrieveSuccessorMsg`[75] message to s_i. s_i replies to this message by sending its successor back to n. Normally the successor will be the $i+1^{th}$ successor in the list. If not, node n believes that s knows its successor better than itself. Node n inserts the new successor of s in the list and checks this successor in the next phase. If it reaches the end of the list, the next phase will begin with the first successor (actually, the next direct successor). The updating of the predecessor list can be done in the same way. Each node maintains a certain number of direct successors and predecessors. The value is defined by the variable `STORED_NEIGHBORS` in the `ChordConfiguration`[76] class and can be changed at run time.

More evaluation of the influence of the number of stored neighbors and update frequency on the network performance can be found in [20]. In our implementation, we set up the default value by the best value, which was introduced in [20].

Updating a finger

Every node regularly runs the `UpdateFingerPointOperation`[77] operation to refresh its i^{th} finger. This operation is not different then starting a lookup request to find the successor of the key to which the i^{th} finger refers. When a node receives the successor, it checks whether the i^{th} finger is also the correct $i+1^{th}$ finger. It happens when the $i+1^{th}$ finger point is between the node and the successor. In this case, the $i+1^{th}$ finger must not be updated and one update operation is saved. On the other side, if all fingers in the finger table are different, only one finger will be updated in each phase. However, according to [29], the authors concluded that the number of distinct nodes in finger table is normally $O(logN)$. The performance thus will be much more improved.

3.3.5.3 Configuring Chord within the simulator

The factory class to configure scenarios using Chord as overlay is `ChordNodeFactory`[78]. Listing 3.41 shows a code snippet of the configuration file, which sketches the denomination of the factory class for Chord.

```
1 <Overlay class="de.tud.kom.p2psim.impl.overlay.dht.chord2.components.
2     ChordNodeFactory"/>
```

Listing 3.41: Configuration of the overlay node as a Chord node

The factory does not provide any methods to change parameters of the overlay. Instead, the parameter values are contained in `ChordConfiguration`[79]. In the following, the parameters are listed and a brief description is given:

- `TRANSPORT_PROTOCOL` - Transport protocol, which should be used in the overlay
- `MESSAGE_TIMEOUT` - If no reply is received in this period, a messages loss is assumable
- `MESSAGE_RESEND` - If no reply is received, the message will be resent a certain amount of times

[75] `de.tud.kom.p2psim.impl.overlay.dht.chord2.messages`
[76] `de.tud.kom.p2psim.impl.overlay.dht.chord2.components`
[77] `de.tud.kom.p2psim.impl.overlay.dht.chord2.operations`
[78] `de.tud.kom.p2psim.impl.overlay.dht.chord2.components`
[79] `de.tud.kom.p2psim.impl.overlay.dht.chord2.components.ChordConfiguration`

- OPERATION_REDO - If an operation failed, the operation will be executed again a certain amount of times
- LOOKUP_TIMEOUT - The time for performing a lookup operation
- UPDATE_SUCCESSOR_INTERVAL - The interval between two operations, which refresh the next direct successor
- UPDATE_FINGERTABLE_INTERVAL - The interval between two operations, which refresh a finger
- UPDATE_NEIGHBORS_INTERVAL - The interval between two operations, which refresh a successor/predecessor in the successors/predecessors list
- STORED_NEIGHBORS - Number of stored successors/predecessors in the list, which are used as backup for a direct successor/predecessor
- MAX_HOP_COUNT - Message will be dropped if the hop count exceeds this value

The second part concerns the configuration of the evaluation components. Listing 3.42 shows a code snippet to integrate the evaluation components into the monitoring-element of the configuration file (we refer to Subsection 5.1.1 for details about the configuration in the configuration file, while Subsection explains the details for combining written Analyzers with the Monitoring Architecture). Within the second Analyzer-element the parameter scheduleTime denotes the simulation time, in which the overlay will be periodically evaluated and the results will be written to an output file. Each component associates with a Boolean variable, which measurements for statistics are activated. Every component is implemented in a class, which extends the abstract class AbstractMetricStore[80]. In the following, we will list the associated parameters of the component ChordOverlayAnalyzer in the configuration file. All implementing classes can be found in the metric-package[81].

```
1 <Analyzer class="de.tud.kom.p2psim.impl.overlay.dht.chord2.analyzer.metric.
2     MessageCounter"/>
3
4 <Analyzer class="de.tud.kom.p2psim.impl.overlay.dht.chord2.analyzer.
5     ChordOverlayAnalyzer" scheduleTime="10m" lookupStats = "true"
6     messageStats = "true" stabilizeStats = "true" peerStats = "true" />
```

Listing 3.42: Configuration of the evaluation for Chord overlay.

- MessageStore (refers to variable messageStats) - Collects sent and received messages in the complete system and evaluates the network traffic. MessageCounter must be registered as a further Analyzer-element within the configuration file to observe the communication between the peers in the overlay
- LookupStore (refers to variable lookupStats) - Counts the number of started lookup requests and evaluates the results
- PeerStore (refers to variable peerStats) - Retrieves the list of participants in the network. This component provides the number of joined and left nodes in a certain period of time
- StabilizeEvaluator (refers to variable stabilizeStats) - Evaluates the stabilization of the Chord overlay. The stability is denoted by the ratio of correct and incorrect direct neighbors as well as its fingers

3.3.5.4 The API of Chord for the simulator

This subsection describes the main functionality of the Chord overlay, which is implemented in the class ChordNode[82]. The class ChordNode provides the methods to other layers above, while the methods can be called directly from the action file as well.

[80] de.tud.kom.p2psim.impl.overlay.dht.chord2.analyzer.metric
[81] de.tud.kom.p2psim.impl.overlay.dht.chord2.analyzer
[82] de.tud.kom.p2psim.impl.overlay.dht.chord2.components

- `join(OperationCallback callback)` - This method starts the join operation to connect a new peer to the overlay. The operation callback will be called when the join process finished.
- `leave(OperationCallback callback)` - This method disconnects a peer from the overlay. As before, the operation callback will be called when the leave process finished.
- `overlayNodeLookup(ChordID key, OperationCallback<ChordContact> callback)` - This method finds the node, which is responsible for the given overlay ID. The operation callback will be executed when the responder is found. The responsible peer is represented by a `ChordContact`-object[83]. The lookup requester can use the transport information within `ChordContact` to directly contact the responsible node.

3.3.6 Kademlia

The following paragraphs of this subsection contain a rough description of the overlay network Kademlia[24] and its basic concepts. The following subsections deal with the implementation of Kademlia within the simulator (Subsection 3.3.6.1), the configuration (Subsection 3.3.6.2), and how it may be used for simulations (Subsection 3.3.6.3).

Kademlia belongs to the family of distributed hash-tables (DHT). Therefore, its main purpose is to efficiently store and lookup key-value pairs distributed among all peers of the system. To form the overlay network each peer has an unique overlay identifier (overlay-ID), represented as 160-bit number. The ID is generated randomly or by hashing the peer's IP-address. Furthermore, data items stored within the DHT are also assigned to IDs, which are generated by hashing the item's content. They are situated in the same key space as the overlay-IDs. A Peer is responsible for storing data items whose IDs are next to its own overlay-ID using the XOR-operator as distance metric. This metric is symmetric and implies that two keys are close to each other if they share a long prefix.

Topology and routing table

The topology of the overlay network always has the form of a binary tree in which peers are placed based on their overlay-IDs. Peers that share a common ID-prefix are members of a common subtree. The so formed binary tree is the basis for the routing table of each peer. Since Kademlia is meant to be used by a huge number of peers, the routing table of a single peer contains only a small subset of the tree. The idea is to only maintain a limited number of contacts of peers that are members of subtrees, which the peer itself is not a member of. That means each such subtree forms a leaf in the locally maintained tree. The leafs are called k-buckets, where k is a configuration parameter that limits the number of peers within a leaf to at most k entries. As a result the size of the maintained tree and the routing table is limited through this structure to O(log(N)), where N is the total number of peers in the network.

Routing

Routing within this overlay is an iterative operation. In general, to send a message to a peer responsible for a given key, the peer can look up another peer in its local routing table that is the nearest to the key and send a request to it. Because of the routing table structure the receiving peer then lies in the right subtree and therefore has more information to find a responsible node. This peer can again look up the nearest peer in his local routing table and send the contact information back to the requesting peer. The requesting peer then can repeat the look up procedure with the new gained knowledge. In this way the requesting peer gets closer and closer to a responsible peer by each request iteration. This is the base mechanism which is further optimized in its realization. To speed it up, the requesting peers do not only send one request at a time but multiple requests in parallel, limited to a maximum by the configuration parameter `alpha`. Furthermore, a peer that receives a request sends back a list with up to k closer peers, not only one. The responsible nodes are then found, if finally all nearer contacts have answered the

[83] `de.tud.kom.p2psim.impl.overlay.dht.chord2.components`

requests and their answers did not contain any new contacts with smaller distances to the key looked up.

Store and lookup

Given the description of the routing mechanism above, a store operation for a given data-item results in a lookup for the item's key by the peer that wants to store it. The lookup finishes with a set of peer contacts that are near to this key. The peer contacts the `k` closest of them, which then locally store the data-item.

A lookup operation for a given key works very similar to store. The key is again iteratively looked up until either a peer is found that holds the searched data-item or the lookup terminates, which means none of the peers holds the item.

Maintenance of the routing table

A peer updates its local routing table when it gets to know new peers during a lookup started by itself or when it has a contact of a peer that sends a query. If there was no update to a bucket for a certain time, the peer starts a lookup for the `k` closest neighbors of a random key that falls into that bucket. In this way, it updates the bucket and announces its existence to all peers contacted during the lookup.

Republishing of data items

Since peers leave and join the network the set of the `k` closest neighbors of a key tend to change over time. To avoid that existing key-value pairs cannot be found or even get lost, the original paper [24] describes a republishing mechanism. After a certain time interval, a peer holding a key-value pair looks up the `k` closest neighbors of the key and assures that they hold the pair. If one peer doesn't already hold the pair, its storage is initiated. In this way, new peers get to know about older data-items they are responsible for. To do the republishing efficiently without wasting to much bandwidth, a peer only republishes a pair if it did not recognize a republishing of the same pair by another peer for a certain time interval.

Join and stabilization

To join the overlay, a peer needs to know at least one peer that is already member of the overlay. It then can insert this peer in its routing table and start a lookup for its own ID. After that, the peer has a good knowledge of its neighborhood. To additionally provide the routing table with contacts further away, the peer refreshes all buckets of its routing table that have a bigger distance than the nearest known neighbor. Nearer buckets must be empty since the look up for the local ID did not discover any nearer peers. The procedure of refreshing has the positive side-effect that the existence of the node is announced to other peers, especially to its direct neighbors.

Leaving of a Host

There is no special mechanism to leave the network. Data items are always replicated on `k` peers and have to be republished after a certain time interval. In this way, there is simply no need for a special action upon the leaving of a node. References to the node get replaced when buckets are refreshed and data items get distributed to other nodes when a republishing takes place.

3.3.6.1 Implementation within the simulator

The implementation of this overlay can be found in the package `kademlia2`[84]. It is the result of a student's bachelor thesis and includes some extra concepts in addition to pure Kademlia as described in the original paper. The thesis includes a comparison of Kademlia to Kandy[11] and an own improvement of Kademlia

[84] `de.tud.kom.p2psim.impl.overlay.dht`

by introducing a virtual hierarchy called HKademlia[22]. We do not describe the implementation of Kandy and HKademlia in this section although its implementation is part of the named package, but refer to the thesis for more details about it. The following paragraphs explain some core concepts of the implementation and are meant to give an overview for other developers. If not mentioned otherwise in the rest of this subsection, the implementing classes can found in the `operations`-package[85].

routing table

The data structure, used to maintain routing tables can be found in the package `routingtable`[86]. It is implemented as a tree, as described in the original paper [24]. Operations to traverse the tree are implemented using the Visitor design pattern[10].

Store and lookup

Store and lookup are implemented as operations[87] within the simulator. A peer creates them via the factory `OperationFactory`.

To store a data item, an instance of `StoreOperatio` is retrieved over the method `getStoreOperation`. When this operation is executed, a lookup of the `k` closest peers around the data key is performed and the storage of the data is directly initiated at this peers.

To look up a data item, an instance of `DataLookupOperation`[88] is retrieved from the factory over the method `getDataLookupOperation`. To allow interchangeability the lookup logic is encapsulated in an extra class implementing the interface `LookupCoordinator`[89]. This class keeps track of the yet not queried contacts and determines which contact has to be queried next. The number of concurrent queries, which is denoted as `alpha`, can be changed as described in Subsection 3.3.6.2.

Maintenance of the routing table and republishing of data items

Both, the maintenance of the routing table and the republishing of data items, are done by using periodica operations. The class `PeriodicOperation` is used for that purpose. During the joining phase of a peer, these periodic operations are created and scheduled. The intervals between refreshing and republishing can be configured as described in Subsection 3.3.6.2.

Joining and stabilization

Joining is initiated by a call to the method `doConnect()` within the class `AbstractKademliaNode`[90]. A call of this method results in a few internal steps. First the peer is marked to be joining, then a `BucketRefreshOperation` is created and scheduled to update already existing routing table entries. After that, a `KClosestNodesLookupOperation` is created and scheduled to look up the peer's own ID and therewith announce the existence of the peer to direct neighbors. As last step, the peer is marked as present and the periodic operations to refresh buckets and republish data items are triggered.

Leaving of a Host

Leaving of a Host is triggered through a call of the method `disconnect()` within the class `AbstractKademliaNode`. This marks the peer as absent and aborts all running operations started by the peer.

[85] `de.tud.kom.p2psim.impl.overlay.dht.kademlia2`
[86] `de.tud.kom.p2psim.impl.overlay.dht.kademlia2.components`
[87] `de.tud.kom.p2psim.api.common.Operation`
[88] `de.tud.kom.p2psim.impl.overlay.dht.kademlia2.operations.lookup`
[89] `de.tud.kom.p2psim.impl.overlay.dht.kademlia2.operations.lookup`
[90] `de.tud.kom.p2psim.impl.overlay.dht.kademlia2.components`

3.3.6.2 Configuring Kademlia within the simulator

The factory class to configure scenarios containing Kademlia as overlay is `KademliaNodeFactory`[91]. Listing 3.43 shows an extract of an configuration file that uses this factory to configure the Kademlia nodes for a simulation.

```
1 <NodeFactory
    class="de.tud.kom.p2psim.impl.overlay.dht.kademlia2.setup.KademliaNodeFactory"
    />
```

Listing 3.43: Configuration of the overlay node - Using Kademlia

The factory does not provide any methods to change parameters of the overlay. The whole configuration is done through an extra file[92]. Within this file the following parameters can be specified:

- `CLUSTER_MAPPING_FILE` - Properties file for hierarchical Kademlia. It contains mappings from group identifiers to cluster suffixes.
- `ID_LENGTH` - The bit length of the Kademlia overlay identifiers.
- `BTREE` - The order of the routing tree. (Each node can have up to 2^{BTREE} children.) In the Kademlia paper, this parameter is called `b`.
- `K` - The maximum number of contacts per bucket.
- `REPLACEMENT_CACHE_SIZE` - The maximum number of contacts per replacement cache.
- `STALE_COUNTER` - The number of marks that allow the routing table to remove an entry of a contact that has not responded to messages. That is, if the contact has failed to respond to queries `STALE_COUNTER` times, it may be dropped.
- `REFRESH_INTERVAL` - The interval in which the routing table buckets are refreshed. The unit is simulation hours.
- `HIERARCHY_DEPTH` - Height of the hierarchy tree. 0 disables the hierarchy. Each step of the hierarchy consumes `HIERARCHY_BTREE` bits in the node identifier.
- `HIERARCHY_BTREE` - Number of branching bits per hierarchy step. That is, each node (cluster) in the hierarchy tree has $2^{HIERARCHY\_BTREE}$ children.
- `INITIAL_ROUTING_TABLE_CONTACTS` - The number of `KademliaOverlayContacts` that will be given to the nodes to initially fill their routing tables.
- `ALPHA` - The maximum number of parallel messages during one node lookup.
- `LOOKUP_MESSAGE_TIMEOUT` - If no lookup reply is received from a node in this period (given in simulated seconds), consider the node as not responding.
- `LOOKUP_OPERATION_TIMEOUT` - Maximum number of simulated seconds for performing a lookup operation.
- `PERIODIC_LOOKUP_INTERVAL` - The interval between two random data lookup operations executed on one Host in simulated minutes.
- `DATA_SIZE` - The assumed size of a data item (in bytes) that is transferred at the overlay (application) layer.
- `DATA_EXPIRATION_TIME` - The time after which data items are evicted from a node's local database (if they have not been republished since then) in simulated hours.
- `DATA_ITEMS` - The number of data items that are available in the overlay network.
- `REPUBLISH_INTERVAL` - The interval in which data items have to be republished in hours.
- `PEERS` - (An approximate of) The total number of peers that are members of the overlay network. This parameter is used as initial size for some collections. The value does not change the behavior of the overlay.

[91] `de.tud.kom.p2psim.impl.overlay.dht.kademlia2.setup.KademliaNodeFactory`
[92] `config/kademlia.properties`

3.3.6.3 The API of Kademlia for the simulator

Methods to trigger operations of Kademlia are defined within the class `AbstractKademliaNode`[93]. That means this class has to be referred to as `componentClass` when defining the `Scenario` part of the configuration file as described in Subsection 5.1.1.

`AbstractKademliaNode` defines the following methods:

- `connect()` - Connects this node to the Kademlia overlay network.
- `disconnect()` - Disconnects this node from the Kademlia overlay network.
- `store(final String key, final String data)` - Stores the given data in the Kademlia overlay network. The data is represented by a String.
- `store(final String key, final String data, OperationCallback callback)` - Same functionality as before but additionally an instance of `OperationCallback`[94] is passed as callback handler.
- `beginLookups()` - Begins periodic lookups for random keys (The lookups are only executed if the node is marked as present).
- `storeRank(final int rank, final String data, OperationCallback callback)` - Stores a Kademlia key and its data with the specified rank.
- `lookupRank(int rank)` - Starts a single lookup for a rank.
- `lookupRank(int rank, OperationCallback callback)` - Same functionality as before but additionally an instance of `OperationCallback` is passed as callback handler.
- `valueLookup(OverlayKey key, OperationCallback<DHTObject> callback)` - Starts a lookup for an object that belongs to the provided `key`.

3.3.7 VON - Voronoi based overlay

The following paragraphs of this subsection contain a rough description of the overlay network VON[13] and its basic concepts. The further subsections deal with the implementation of the overlay within the simulator, the configuration and how it may be used for simulations.

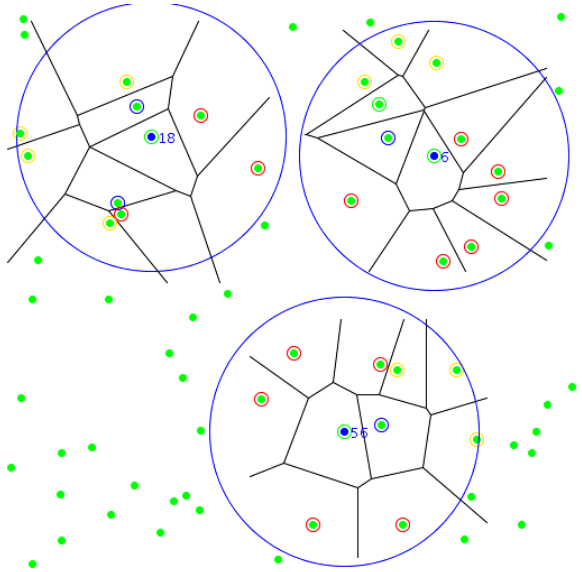
The VON overlay belongs to the family of Information Dissemination Overlays (IDO). It is designed to provide a highly scalable P2P overlay structure for networked virtual environments [19], such as multi-player online games. Fundamental to such virtual environments is a large number of users, with each user corresponding to a virtual entity in the environment, such as a game character. Each entity features a position on the world map of the virtual environment and may move through this map as a result of user actions. Given that all entities may move through the world, the focus of the IDO is to give each entity a consistent knowledge about other entities nearby their position to enable an efficient direct communication between them. This is very essential as nearby entities might have to be directly informed about important user actions without high delays. The need for this becomes clear when considering the most prominent examples, online multi-player games, where actions of one player may directly influence the state of other players, e.g. during a fight between the entities of players. While the direct communication to neighbors is the most important goal of an IDO, it is also important that an entity does not know too many neighbors. This would result in a system that is not scalable, as too many connections would have to be maintained and unnecessary network load would be generated due to messages being sent to neighbors that are not interested in the events of the sender.

Topology and neighbors

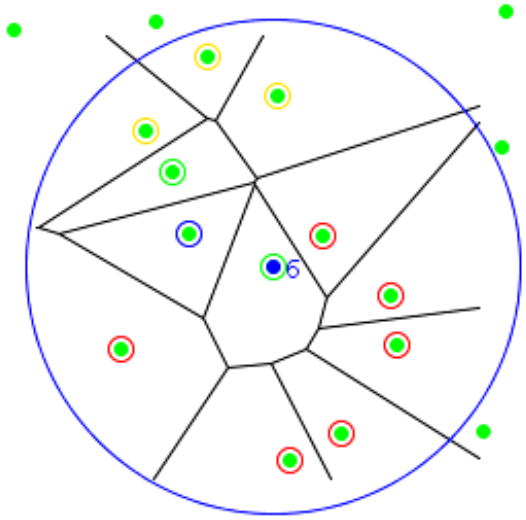
To provide the described functionality, VON partitions the space of the virtual world by using Voronoi diagrams [5]. This is a mathematical concept from the field of geometry, which allows to completely

[93] `de.tud.kom.p2psim.impl.overlay.dht.kademlia2.components.AbstractKademliaNode`
[94] `de.tud.kom.p2psim.api.common.OperationCallback`

decompose a space into non-overlapping regions. Each of these regions is enclosing a single point from a given finite set of points placed in the space. Simply spoken, the region belonging to a given point is defined by the part of the space, which is closer to this point than to any other point of the set. In VON the set of points is derived from the positions of the nodes that are part of the overlay. Each node corresponds to a point in the global Voronoi diagram. However, the idea of a global Voronoi diagram is just conceptual to clarify the basic idea. To make the system scalable, the knowledge a node has about the overall system must be limited and should include only a small subset of the nodes in the system (see Figure 3.14a for a visual representation). This is realized by the concept of a so called area-of-interest (AOI). The AOI is simply defined by a radius, which describes a circle around a node. All other nodes that are inside this circle are called AOI neighbors and have to be known by the node in the center. In addition to this, a node has to know all other nodes whose regions directly enclose its own region. These neighbors may already be included in the set of AOI neighbors, but can also lie outside the AOI circle. This way it is ensured that the overlay is kept connected, even if there are parts of the virtual world that only contain a small number of nodes. Figure 3.14 shows the basic topology and the different types of neighbors.



(a) The topology of VON, the local Voronoi diagram of three nodes is shown.



(b) The different types of neighbors.

Figure 3.14: The topology of VON and the types of neighbors. Boundary neighbors are marked orange, enclosing neighbors blue, normal AOI neighbors green. If an enclosing neighbor also is a boundary neighbor, it is marked red.

Joining of peers

Before discussing the joining procedure, it is important to mention that VON was designed under the assumption that there is a provider that maintains at least one node in the system. This node is well-known and responsible for assigning the overlay-ID to new nodes and may be used as bootstrap node to join the overlay. The assignment of a valid overlay-ID always has to take place before the joining of a node.

The joining then consists of the following steps: A node chooses an initial position to join the virtual world. It then sends a joining request to a node, which is already part of the overlay. This is usually the bootstrap node mentioned before. The first contacted node checks its local Voronoi map, if the position of the joining node lies inside its own Voronoi region. If this is not the case, it chooses the node within its local Voronoi diagram that is the nearest to the joining nodes position and forwards the joining request. This forwarding is called a greedy forward mechanism, as the distance to the destination gets smaller

in each step. Finally, the request will arrive at the node that is responsible for the new node's position. This node then informs the joining node about all neighbors that it should be aware of, according to its local knowledge. The joining node then contacts each of these nodes and announces its existence to them. During this first contact, the joining node tells each of the node about its already known neighbors. This is important, so the contacted node can check if the node is lacking neighbors. If there are missing neighbors, the contacted node informs the joining node about them. After the joining node contacted all the neighbors it got to know during this procedure, it is fully connected to the overlay.

Movement of peers

As mentioned before, the nodes may move and thereby change their position in the virtual environment. For a consistent overlay topology it is essential that a node informs its neighbors about new positions. A new position can have consequences on the neighborhood of the node itself as well as on the neighborhood of other nodes. A node may leave or enter the AOI of an other node due to its movement. For this reason a node always sends a so called move-message to all of its neighbors after changing the position and then updates its own local Voronoi diagram accordingly. Nodes, that are not relevant anymore are simply removed from the local neighbors. To learn about new neighbors, a node depends on its so called boundary-neighbors (see Figure 3.14b, boundary neighbors are marked orange). These are the neighbors inside the AOI whose regions intersect with the boundary of the AOI. If such a neighbor receives a move-message from the node, it checks if the update of the position leads to new nodes entering the AOI. If this is the case, the boundary neighbor informs the node about them. This way the topology is kept consistent, as all nodes know the neighbors they should know including their correct position.

Changes to the specification

During the implementation of VON for the simulator, we encountered some problems that where not sufficiently covered by the VON specification [13]. Therefore this paragraph explains our solutions to these problems.

One problem was the discovering of failed nodes and the handling of lost messages. As VON does not include acknowledgements for sent messages at the overlay level and we used UDP as transport protocol, the sender of a message can never be sure that his message arrived at its destination. But this is necessary to be sure joining requests arrive at their destination during the forwarding procedure. Furthermore, this is essential to discover failed neighbors when trying to send them position updates. To solve this problem, we introduced an hop-for-hop acknowledgement mechanism for the forwarding of the joining request and so called heart-beat messages to detect failed nodes. The acknowledgement mechanism is very simple. It introduces the sending of an ACK-message after a node received a forwarded joining request. If this message does not arrive within a certain amount of time, the sender tries to resend the request again a configurable amount of times. In addition, a joining node also introduces a timeout, after which it considers the joining attempt as failed and re-initiates the whole procedure. This way we can ensure that nodes join the overlay correctly in present of churn and message losses. The introduction of heart-beat messages was implemented to allow the detection of failed neighbors. This concept is not necessary, if we assume that every node is constantly moving and therefore sending messages to all neighbors to inform them about position updates. But we assume that nodes may not move all the time and may keep their position for a while. Due to the limitations of the simulator and the limitation of the original VON specification [13], it is not possible to distinguish between neighbors that failed and those, which just did not move. The idea of the heart-beats is that a node sends update messages to its neighbors, even if it does not move. This is only done after a certain period of time, when the node did not move. The advantage of this concept is that nodes now can keep track of the time, they had the last contact to their neighbors. If a node did not receive an update message from one of its neighbors for a multiple of the maximum time between heart-beats, it may assume that the neighbor failed and just remove him from the local Voronoi diagram. This mechanism worked very well in our simulations including churn.

However, the choice of a good value for the time interval after which a neighbor is considered as failed may be important for further studies.

3.3.7.1 Implementation within the simulator

The implementation of VON can be found in the von-package[95]. It contains all base classes of the overlay and some components for evaluation as well as a visualization of the overlay.

As most of the base classes of the overlay stick to the general structure of an overlay implementation, it does not make sense to describe them in too much detail. Important to mention is the package `voronoi`. This package includes the logic to handle Voronoi diagrams. The base for this code was taken from the code of VAST [14], an open source implementation of VON, published under the GNU Public Licence as denoted in the source files. We adapted the code to our needs but left its core functionality untouched. Another package to mention is `models`. It includes models for positioning and the movement of nodes. To be extensible at this point, the package includes the definition of interfaces for the models as well as sample implementations. To implement new models take a look at both, the interfaces to be implemented as well as at the sample implementations.

The components for evaluating the overlay can be found in the package `evaluation`. The evaluation uses the analyzer architecture of the simulator and can be used for simulations by adding the class `EvaluationControlAnalyzer` as analyzer in the configuration file. Once added as analyzer, the evaluation collects data and computes several metrics after a configurable time interval. Some results are printed on the console, while the majority of the data is written into a file that can be used for plotting with the tool GnuPlot. The computed metrics include overlay specific metrics as well as basic metrics like the bandwidth consumption.

The visualization component is located in the package `visualization`. It only includes one class, namely `VisWindow`. This class can also be added as analyzer in the configuration file. During simulations, it will create and display a `JFrame` that shows the topology of the overlay and allows some interaction, e.g. to select nodes and display their local Voronoi diagrams.

3.3.7.2 Configuring the overlay within the simulator

The configuration of VON is possible via the class `VonConfiguration`[96]. It includes all important parameters, which are described in the following paragraph:

- `GENERAL_MSG_TIMEOUT` - The timeout used for message transmissions
- `GENERAL_MSG_RETRANSMISSIONS` - The maximum number of retransmissions at message timeouts
- `OP_TIMEOUT_OBTAIN_ID` - The timeout for the operation to obtain an ID
- `OP_WAIT_BEFORE_RETRY_OBTAIN_ID` - The time interval to wait before the retry of a failed operation to obtain an ID
- `OP_TIMEOUT_HELLO` - The timeout for the operation to contact new neighbors
- `OP_TIMEOUT_MOVE` - The timeout for the operation that moves a node
- `OP_TIMEOUT_JOIN` - The timeout for the operation to join the overlay
- `OP_WAIT_BEFORE_RETRY_JOIN` - The time interval to wait before the retry of a failed operation to join the overlay
- `WORLD_DIMENSION_X` - The size of the world in dimension X
- `WORLD_DIMENSION_Y` - The size of the world in dimension Y
- `DEFAULT_AOI_RADIUS` - The default AOI radius to be used on initialization of a node
- `TRANSPORT_PROTOCOL` - The used transport protocol

[95] `de.tud.kom.p2psim.impl.overlay`
[96] `de.tud.kom.p2psim.impl.overlay.von`

- POSITION_DISTRIBUTION - An instance of IPositionDistribution, which defines the distribution of node positions
- MOVE_MODEL - An instance of IMoveModel, which defines the movement model used by the nodes
- MOVE_SPEED_LIMIT - A speed limit that defines, which maximum distance a node may move during one movement step
- MOVE_TIME_BETWEEN_STEPS - The time interval between movements of a node
- INTERVAL_BETWEEN_HEARTBEATS - The maximum time after which a heart-beat is done if no movements took place
- STALE_NEIGHBOR_INTERVAL - The time interval after which a neighbor is considered stale and may be removed from the local Voronoi diagram
- INTERVAL_BETWEEN_STATISTIC_GENERATIONS - The time interval between the statistic generations
- STATISTICS_OUTPUT_PATH - The path to be used as output for statistic data

3.3.7.3 The API of the overlay for the simulator

Methods to trigger operations within VON are defined in the class VonNode[97]. That means this class has to be referred to as componentClass when defining the Scenario part of the configuration file as described in Subsection 5.1.1.

VonNode defines the following methods:

- join() - Initialize the joining to the overlay.
- leave() - Initialize the leaving of the overlay.

In addition, there are four more methods that can be used with the movement model VariableSpeedMoveModel[98]:

- joinAndMove() - Initialize the joining and instantly start moving.
- startDecreaseSpeed() - Begin to decrease the speed until a complete stop.
- startIncreaseSpeed() - Begin to increase the speed until the speed limit is reached.
- stopSpeedChanging() - Stop the changing of speed and keep the current speed.

An application that is built on top of this overlay can access the collection of current neighbors via the method getCurrentOverlayNeighbors().

[97] de.tud.kom.p2psim.impl.overlay.von.VonNode
[98] de.tud.kom.p2psim.impl.overlay.von.models.

3.4 Application Layer

This section details the Application Layer of PeerfactSim.KOM. Due to the fact, that applications of a P2P system, including their offered functionality, can significantly vary, the simulator only provides a sparse interface called `Application`[99] and a skeletal implementation `AbstractApplication`[100]. The interface provides two methods to start and stop the application, while the abstract class contains the integration of the application into the simulator. These two elements constitute the basis to implement new applications, while no interface or abstract class for the messages or their exchange is provided.

3.4.1 Filesharing2

Currently, PeerfactSim.KOM only consists of one implementation for the Application Layer, which is called Filesharing2 and models a simple application on top of any search overlay. The implementation of Filesharing2 can be found in the `Filesharing2`-package[101]. Filesharing2 allows to define large sets of documents along with the distribution of their lookup and publish probability. Afterwards, the application randomly takes documents to store and publish them, according to the defined probability in the set. To store and publish the documents, the application relies on the offered functionality of the overlay. Based on the published documents, the peers can periodically seek for a document, where the time between to consecutive search operations is exponentially distributed. In contrast to the periodic lookup, the application enables the execution of a single lookup for a randomly chosen document or for a set of documents based on the provided keys. A detailed description of the implemented operations, which are included in the application, are described in Subsection 3.4.1.2.

3.4.1.1 Definition of document sets

For simulating the application, we have to define the document sets in the beginning. Some simple actions of Filesharing2 do not require need any document set, but we recommend to use them. Therefore, we add the element `ResourceSpace` to the configuration file as described in Listing **??**. Inside this element, we can define multiple resource sets. Every resource set has a name, which becomes important when we will use it in the actions file later on. In addition, the set is characterized be the `size`-attributed, which, for example, specifies the number of resources contained in this set.

Currently there are three document set implementations:

- `ZipfDocumentSet`: This document set contains documents that are stored and looked up with a Zipfian-distributed probability. So there are a few documents that are very popular and will be often stored and looked up, while many other documents are only looked up occasionally. This document set models the Zipf distribution, which in 2003 represented the popularity distribution for most resources in the web, as stated by Gummadi et al. [12]. But they also discoverd that this distribution did not necessarily reflect the popularity of content in P2P filesharing applications. Other measurement studies, such as [8] in 2010 on BitTorrent systems, supported this result, while, e.g. Lloret et al.[23] discovered in 2004 that the popularity of content in P2P filesharing applications, however, can follow Zipf distribution. Examples of Zipf distributions are depicted in Figure 3.15. To make it even more realistic, documents are occasionally reordered in the Zipfian distribution rank by swapping two random documents in the set. Sometimes, documents suddenly become more popular or will lose their popularity. The parameter `meanReorderIntvl` represents the interval of this reordering process.

[99] `de.tud.kom.p2psim.api.application`
[100] `de.tud.kom.p2psim.impl.application`
[101] `de.tud.kom.p2psim.impl.application`

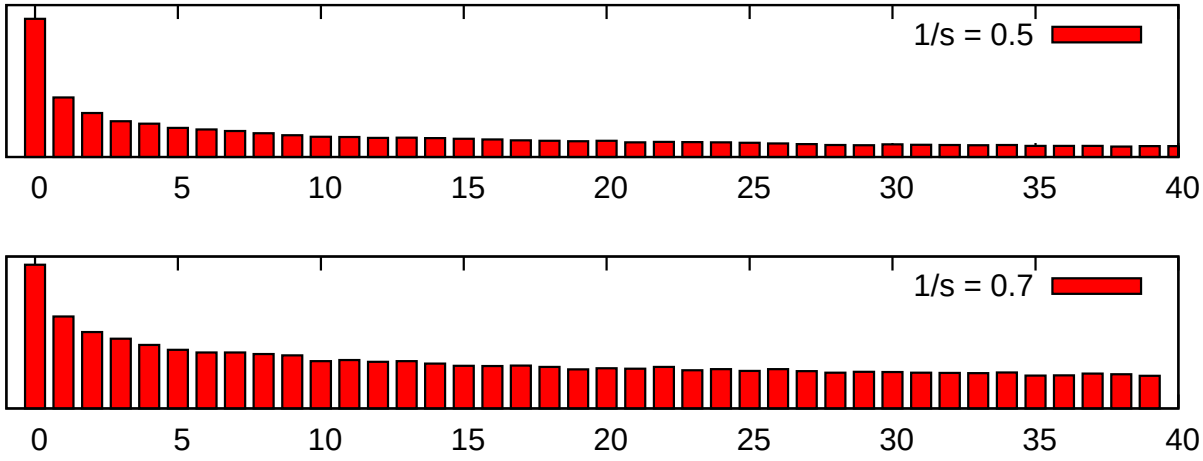


Figure 3.15: Example of Zipf distributions with two different 1/s parameters

- `FlatDocumentSet`: Every document of this set has the same probability to be stored and looked up.
- `UniqueDocumentSet`: In this set, a document may only be stored *once* by one peer. If all documents of this set were already stored and there is an attempt to store one more document from this set, an exception is thrown. So define the size of this document large enough. The lookup probability is equally distributed like in the FlatDocumentSet. This kind of document set models common behavior e.g. in *instant messaging* scenarios, where every peer publishes its contact data, which is unique.

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <Configuration>
3  <!--   ...   -->
4      <ResourceSpace
           class="de.tud.kom.p2psim.impl.application.filesharing2.documents.\
5    DocumentSpace" static="getInstance">
6    <ResourceSet class="de.tud.kom.p2psim.impl.application.filesharing2.documents.\
7           ZipfDocumentSet" name="files1" size="$size" zipfExp="0.7"
               meanReorderIntvl="10m"/>
8    <ResourceSet class="de.tud.kom.p2psim.impl.application.filesharing2.documents.\
9           FlatDocumentSet" name="files2" size="150"/>
10   <ResourceSet class="de.tud.kom.p2psim.impl.application.filesharing2.documents.\
11          UniqueDocumentSet" name="files3" size="500"/>
12     </ResourceSpace>
13 <!--   ...   -->
14 </Configuration>
```

Listing 3.44: Document set configuration

3.4.1.2 Actions

This subsection details the offered operations of Filesharing2. The operations can be included in the code of other components of the simulator, or can be integrated in an action file to define a scenario. The action file then specifies when which action should be executed by which peer or group of peers (we refer to Section 5.1.3 for details about the action file).

- **join**: Lets a peer join the overlay, on which this application is running

- **leave**: Lets a peer leave the overlay, on which this application layer is running
- **publishResourcesFromSet <setName> <meanAmount>**: Publishes a random amount of resources from the set given in *setName*. The random amount of resources is exponentially distributed, where *meanAmount* is the mean value of this exponential distribution.
- **publishOneResourceFromSet <setName>**: Publishes exactly one resource from the set given in *setName*
- **publishResources <resources>**: Publishes the resources given with their IDs in a comma-separated string in the parameter *resources*. Example for the parameter *resources*: “1,2,3,4,5”.
- **lookupResourceFromSet <setName>**: Looks up a random resource from the set *setName*. Only takes resources that were published before. If no resource of this set was published before, an exception is thrown.
- **lookupResourceFromSetPeriodically <setName> <meanPeriod>**: Periodically looks up a resource from the set *setName*. Only takes resources that were published before. If no resource of this set was published before, an exception is thrown. The lookup period is exponentially distributed with the mean value *meanPeriod*
- **lookupResourcePeriodically <documentID> <meanPeriod>** Periodically looks up the resource with the rank *documentID*. The lookup period is exponentially distributed with the mean value *meanPeriod*.

4 Monitoring and Analyzers

To do well-founded evaluations, it is crucial to have access to all relevant data a simulation produces, as stated by Naicken et al. [25]. Depending on the focus of a simulation, this data may be generated at any simulated Host and all its layers. It is not possible to generally predict all kinds of data that might be important for the purpose of evaluation. Hence, PeerfactSim.KOM does not provide a mechanism that is capable to collect all possible types of data. Nevertheless, there is a Monitoring Architecture that addresses an easy collection of basic data types. To treat more specialized types of data, individual mechanisms have to be developed.

The following sections describe the Monitoring Architecture of PeerfactSim.KOM in detail. While Section 4.1 explains the general structure and the single components, Section 4.2 elaborates on the concepts of Analyzers, and Section 4.3 gives a practical introduction to the implementation and registration of Analyzers.

4.1 The Monitoring Architecture

The core of the Monitoring Architecture is made up by three elements, the interface `Monitor`[1], its default implementation `DefaultMonitor`[2], and the interface `Analyzer`[3]. Figure 4.1 shows the basic structure of the architecture, including some additional elements of the simulator and their dependencies.

The architecture is a realization of the Observer Design Pattern [10]. In this case, Observers are classes, which are interested in certain event types during a simulation. They implement `Analyzer` or one of its nested interfaces and have to be registered at the simulator. The interface `Monitor` defines the methods for registration, as well as several methods to trigger the notification of registered Analyzers. The main class of the simulation, namely `Simulator`[4], holds a reference to an instance of an implementation of `Monitor` and is used to trigger notifications from any layer. It is configured via the configuration file, which is described in Chapter 5. The default implementation of `Monitor` is `DefaultMonitor`. It maintains a separate list for each type of Analyzer and handles their notification when the appropriate trigger methods are called from components within the simulation. Figure 4.2 shows an overview of the simulator's structure and its dependencies to the monitoring component.

From an abstract point of view, the simulator defines a set of interception points that can be used to collect data during simulations. To use a certain type of interception point, one has to implement the corresponding Analyzer and register its implementation via the configuration file. The simulator notifies the registered Analyzer by calling the methods corresponding to the occurred events. As parameter it passes a set of information about the event. At this point, the information can directly be used for evaluation or only be gathered to do an evaluation at a later point in time.

4.2 The `Analyzer` interfaces

The interface `Analyzer`[5] is the base interface for the realization of all Analyzers, which are situated in the same package. Figure 4.3 shows the Analyzer API, including all available sub-interfaces of `Analyzer`. Even though the single interfaces and their methods should be self-explanatory, we explain them in detail in the following to give a good impression how they could be used to implement Analyzers.

1 `de.tud.kom.p2psim.api.common`
2 `de.tud.kom.p2psim.impl.common`
3 `de.tud.kom.p2psim.api.analyzer`
4 `de.tud.kom.p2psim.impl.simengine.Simulator`
5 `de.tud.kom.p2psim.api.analyzer`

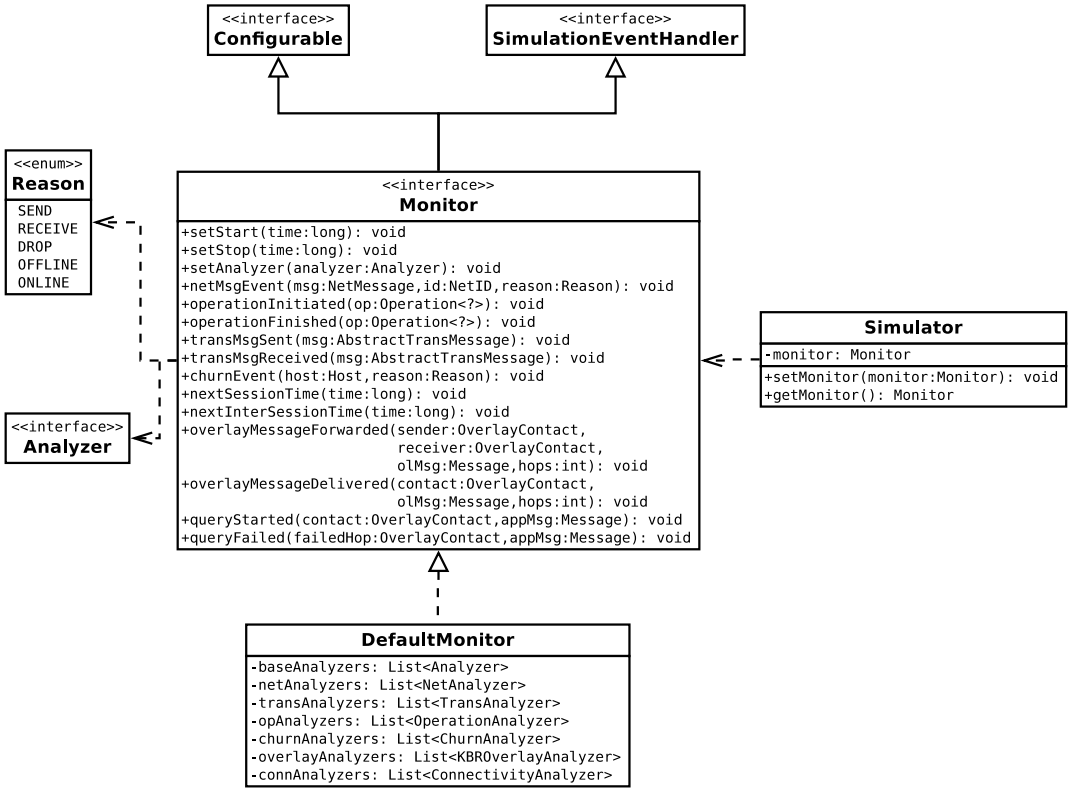


Figure 4.1: The Monitoring Architecture

`Analyzer`: As mentioned before, this is the base interface for all Analyzers. It defines two methods, namely `start()` and `stop(Writer)`. These methods are called by the simulator when the Monitor is started or stopped. The simulation time at which the simulator starts or stops the Monitor is configured in the simulation's configuration file (Listing 4.1 shows an example).

The two methods are usually used to initialize Analyzers and make them ready for the processing and collection of simulation events.

`NetAnalyzer`: This interface defines the methods `netMsgDrop(NetMessage,NetID)`, `netMsgReceive(NetMessage,NetID)`, and `netMsgSend(NetMessage,NetID)`. They are called by the Network Layer upon drop, receive, or send of network layer messages. The parameters provide access to the messages and the Network Layer Id of the Host triggering the event.

Implementations of this interface are usually used to calculate Network Layer statistics, such as the number of sent/received/dropped messages, bandwidth consumptions.

`TransAnalyzer`: This interface defines the methods `transMsgReceived(AbstractTransMessage)` and `transMsgSent(AbstractTransMessage)`. The Transport Layer calls these methods after sending or receiving Transport Layer messages. The parameter is a reference to the message that triggered the event.

Implementations of this interface may be used to generate statistics about end-to-end communications in the network.

`OperationAnalyzer`: This interface defines the methods `operationInitiated(Operation)` and `operationFinished(Operation)`. They are called when operations are initiated, which means they are executed by the simulator, or when operations are finished, which means they end. Ending can be suc-

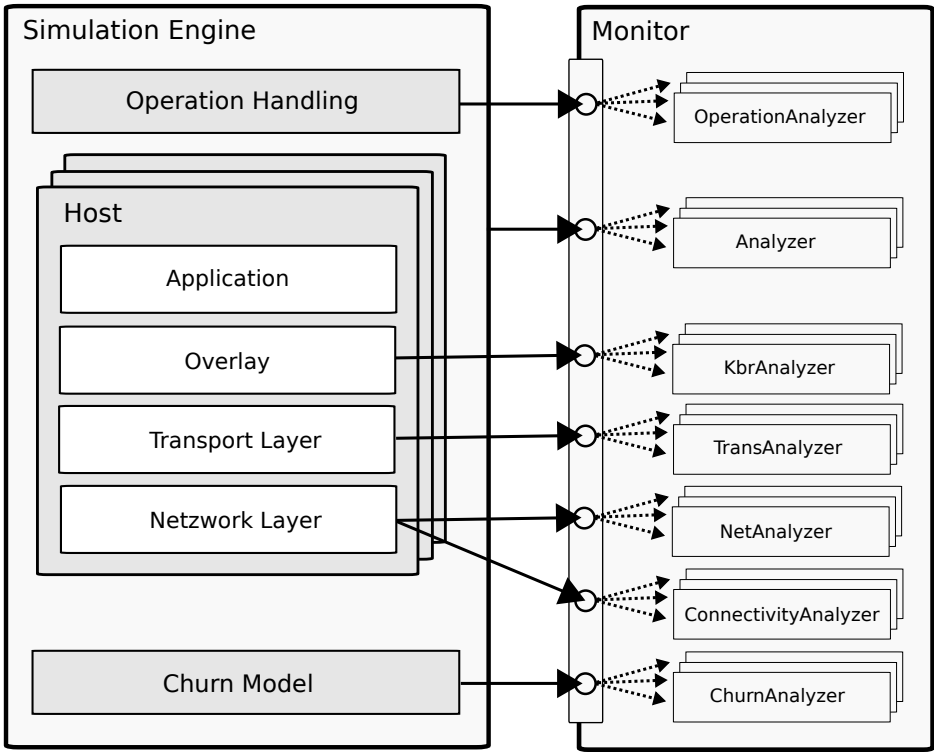


Figure 4.2: The Monitoring Architecture and its dependencies to the simulator

cessful or unsuccessful. In both cases `operationFinished(Operation)` is called. To check whether the operation ended successfully or not, use the methods `isSuccessful()` and `isError()`, which are defined by `AbstractOperation`[6], the base class of all operations. When an operation finishes successfully, the method `getResult()` can be used to retrieve the result of the operation, if there is any.

Implementations of this interface can be used to generate basic statistics, like e.g. the execution time of operations. Furthermore, specialized Analyzers can be realized to gather information about specific operations. These Analyzers are typically bound to a specific overlay or application and include checks for concrete operation classes and casts. Once they are casted, the full API of the concrete operation is accessible and may allow access to more specific information about the operation.

`ConnectivityAnalyzer`: This interface defines the methods `onlineEvent(Host)` and `offlineEvent(Host)`. They are called when the Network Layer of a Host is disconnected or reconnected to the network. A typical example for this is the change of connectivity when simulating churn. In this case, Hosts are disconnected and later on reconnected according to a given churn model.

Implementations of this interface may be used to generate statistics about e.g. the number of Hosts connected to the network or the number of Hosts affected by churn.

`ChurnAnalyzer`: This interface defines the methods `nextInterSessionTime(long)` and `nextSessionTime(long)`. They are called by implementations of `ChurnModel`[7] and give information about the session and inter-session times of churn affected Hosts. This information can be used to estimate the mean and median (inter-)session lengths of the applied churn model.

[6] `de.tud.kom.p2psim.impl.common`
[7] `de.tud.kom.p2psim.api.churn`

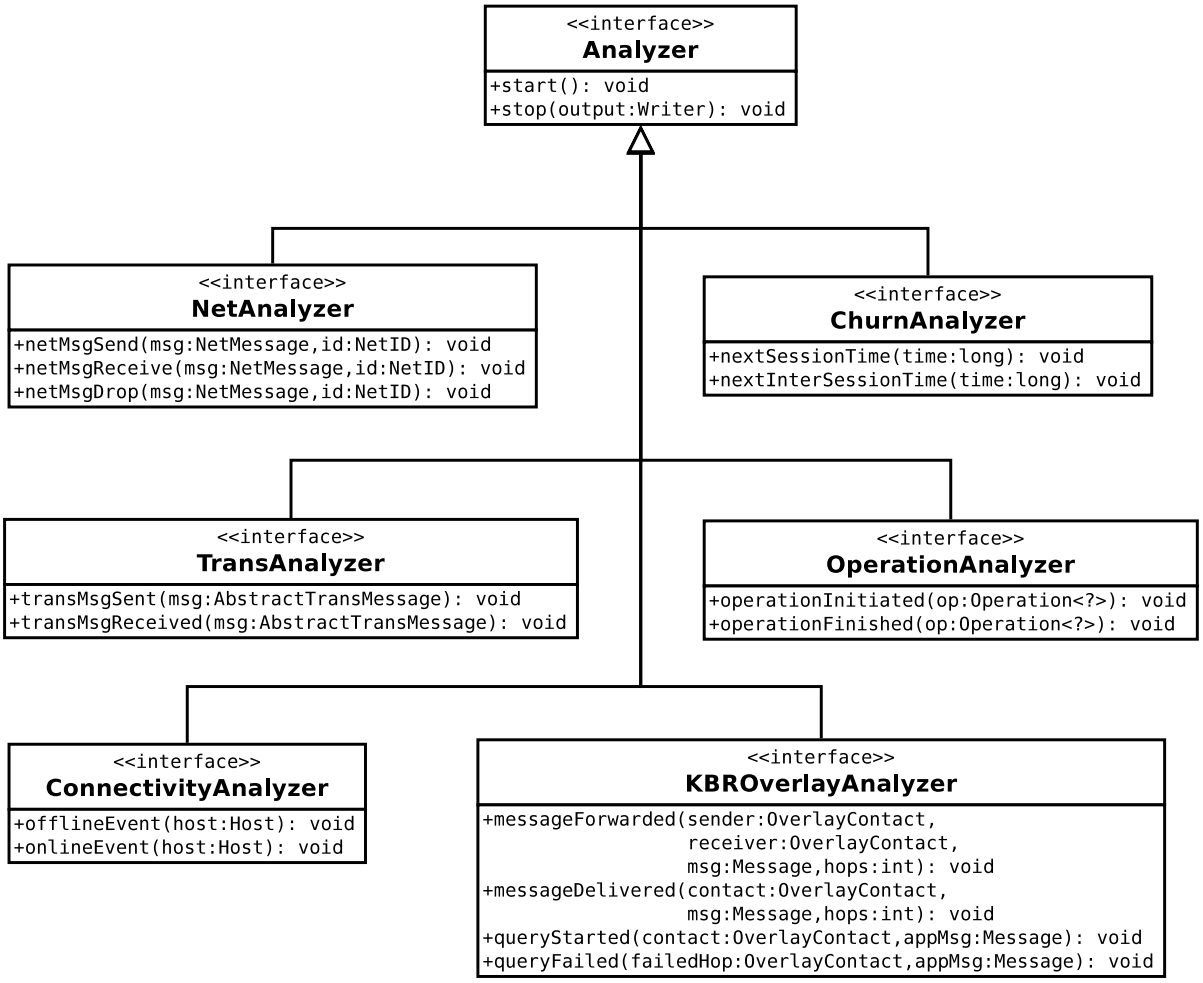


Figure 4.3: The Analyzer API

`KBROverlayAnalyzer`: This interface was defined to allow a uniform evaluation of DHT overlays, which comply to the key-based routing interface (KBR [7]). KBR uses the overlay's functions to realize an abstract way to access the routing capability of the overlay. `KBROverlayAnalyzer` can be used to gather information about this routing process when using an overlay that implements and uses the interface `KBR`[8]. Therefore it defines the following methods:

- `queryStarted(OverlayContact,Message)`
- `queryFailed(OverlayContact,Message)`
- `messageForwarded(OverlayContact,Message,int)`
- `messageDelivered(OverlayContact,Message,int)`

In the context of KBR, a query describes the process of routing a message towards a peer responsible for a given key. This may include several routing hops, where each participating peer either forwards the message towards a peer closer to the key or accepts the message if it is responsible for the key. `queryStarted(...)` is called after a new routing process was started, `queryFailed(...)` when a query did not succeed, `messageForwarded(...)` after a peer forwarded a message, it was not responsible for, and `messageDelivered(...)` after a message was delivered at the responsible peer.

[8] `de.tud.kom.p2psim.api.overlay`

4.3 Implementation and registration of Analyzers

The implementation of Analyzers includes two steps, (1) the implementation of one or several Analyzer interfaces, described in Section 4.2, and (2) the registration of the Analyzer via the configuration files. The configuration is straightforward and follows the principles described in Chapter 5, the chapter about configuring and running simulations. Listing 4.1 shows an simple example of how the configuration of the Monitor and two Analyzers could look like.

```
<Monitor class="de.tud.kom.p2psim.impl.common.DefaultMonitor" start="0"
    stop="30m">
  <Analyzer class="de.tud.kom.p2psim.impl.somepackage.ConcreteAnalyzer1" />
  <Analyzer class="de.tud.kom.p2psim.impl.somepackage.ConcreteAnalyzer2" />
</Monitor>
```

Listing 4.1: Example for the configuration of a simulation's Monitor and its Analyzers.

Implementations of Analyzers are often used to collect a large amount of data. A naive approach for this is to keep the data in main memory and use it for evaluation once the simulation ends. Depending on the types of data collected, the memory usage for it can easily exceed the memory used for the simulation itself. This often becomes a problem when scaling simulations up to tens of thousands of peers, where memory usually becomes the most limiting factor. An approach to solve this issue is to write out the collected data to files, free the memory, and continue the gathering of new data. In this case, the processing of the data is done in a separate, independent step after the simulation finished. The advantage of this approach is that less main memory is occupied and, therefore, larger simulations are feasible.

Another approach is to not only write out raw data but to regularly do a partial evaluation of the collected data, write out the results, free the memory, and continue the collection of new data. While this is not possible for all types of evaluations, it is often a good solution for many scenarios.

Both approaches can be implemented by using Analyzers plus some other mechanism that the simulator defines. Listing 4.2 shows an example of a simple Analyzer to regularly do evaluations during a simulation. In Subsection 4.3.2, we describe a more sophisticated component that can also be used to periodically write out evaluation results.

```
import de.tud.kom.p2psim.api.analyzer.Analyzer;
import de.tud.kom.p2psim.api.simengine.SimulationEventHandler;
import de.tud.kom.p2psim.impl.simengine.SimulationEvent;
import de.tud.kom.p2psim.impl.simengine.Simulator;

public class SomeEvaluationAnalyzer implements Analyzer, SimulationEventHandler {

  private static final long TIME_BETWEEN_STEPS = 5 * Simulator.MINUTE_UNIT;

  @Override
  public void start() {
    doEvaluationStep(); // The first evaluation step
  }

  @Override
  public void stop(Writer output) {
    doEvaluationStep(); // The final evaluation step
  }

  @Override
  public void eventOccurred(SimulationEvent se) {
    doEvaluationStep();
```

```
23    }
24
25    private void doEvaluationStep() {
26      doEvaluation();
27
28      // Schedule the event for the next evaluation step
29      long timeToRedo = Simulator.getCurrentTime() + TIME_BETWEEN_STEPS;
30      Simulator.scheduleEvent(this, timeToRedo, this,
31          SimulationEvent.Type.OPERATION_EXECUTE);
32    }
33
34    private void doEvaluation() {
35      // Do some evaluation
36      ...
37    }
38  }
```

Listing 4.2: A simple Analyzer to regularly do evaluations during a simulation.

In the following subsections, we describe two components that might be helpful for the implementation of own Analyzers for evaluation purposes.

4.3.1 Access to a global view for evaluation

Besides data, collected by using the Analyzer API (described in Section 4.2), it might be necessary to have a global view on a running simulation. For this reason, we implemented the class `GlobalOracle`[9]. It gives access to all existing Hosts of a simulation.

Listing 4.3 shows the component's registration via the configuration file. Once it is registered, it can be accessed statically from all components during a simulation. Nevertheless, there should never be an access to this component other than for evaluation purposes. It provides a view to the system that a single Host normally does not have. Be aware of this fact, when using `GlobalOracle`. Having this in mind, the component allows an easy way to iterate over all available Hosts, retrieve instances of their configured layers and generate statistics. Listing 4.4 shows a simple example how it can be used to count the number of present peers in an overlay.

```
1  <Configuration>
2    ...
3    <HostBuilder class="..." experimentSize="...">
4      ...
5    </HostBuilder>
6
7    <ChurnGenerator class="..." start="...">
8      ...
9    </ChurnGenerator>
10
11   <Oracle class="de.tud.kom.p2psim.impl.util.oracle.GlobalOracle" />
12
13   <Scenario class="..."
14     actionsFile="..."
15     componentClass="..." />
16
17 </Configuration>
```

Listing 4.3: Configuration of `GlobalOracle` to gain a global view on the system.

[9] `de.tud.kom.p2psim.impl.util.oracle`

```
1  ...
2  int numOfPresentPeers = 0;
3
4  for (Host h : GlobalOracle.getHosts()) {
5    OverlayNode n = h.getOverlay(AbstractOverlayNode.class);
6    if(n != null && n instanceof AbstractOverlayNode){
7      AbstractOverlayNode olNode = (AbstractOverlayNode) n;
8      if(olNode.getPeerStatus() == PeerStatus.PRESENT)
9        numOfPresentPeers++;
10   }
11 }
12 ...
```

Listing 4.4: An example how `GloablOracle` can be used to count the number of present overlay nodes.

4.3.2 Periodically write evaluation results to a file

The simulator includes the abstract class `AbstractEvaluationAnalyzer`[10], which can help to easily implement Analyzers to periodically write out statistics to files. It is primarily aimed to write data files for plotting tools, such as `gnuplot`[1]. This abstract class includes all logic for the periodic invocation and file handling. This way, it allows to concentrate on the evaluation logic itself, avoiding redundant code. Listing 4.5 shows a minimal example for its use.

[10] `de.tud.kom.p2psim.impl.analyzer`

```java
package de.tud.kom.p2psim.impl.analyzer;

import de.tud.kom.p2psim.impl.simengine.Simulator;

public class SomeAnalyzer extends AbstractEvaluationAnalyzer {

  public SomeAnalyzer() {
    /*
     * Set options (optional)
     */
    setBeginOfAnalyzing(0);                      // This is the default value
    setEndOfAnalyzing(Simulator.getEndTime()); // This is the default value
    setFlushEveryLine(true);                     // Write lines without buffering
    setFolderName("SomeName");                   // The output folder name
    setOutputFileName("SomeName.dat");           // The output data file name
    setTimeBetweenAnalyzeSteps(5 * Simulator.MINUTE_UNIT); // Default is 1 minute
  }

  @Override
  protected String generateHeadlineForMetrics() {
    // Generate a headline that is written at the beginning of the file
    return "#The headline";
  }

  @Override
  protected String generateEvaluationMetrics() {
    // Calculate metrics and return them as string
    return "value1\t value2\t value3";
  }
}
```

Listing 4.5: A minimal example how `AbstractEvaluationAnalyzer` can be used to periodically write out statistics to a file.

5 Configuring and running a simulation

This chapter deals with the configuration of scenarios for PeerfactSim.KOM as well as with their launching. Therefore, Section 5.1 describes the process of configuration and sketches, which components are needed to create a valid configuration file. Furthermore, it highlights the processing of that file within the simulator and describes the creation of objects for a simulation out of that file. Finally, we complete this section with the introduction of the so-called *action file*, that embodies initial actions for the peers of the scenario. After highlighting the configuration, we introduce how one can start a simulation of the defined scenarios. Therefore, Section 5.2 outlines the currently existing alternatives to start a simulation. On the one hand, the simulator consists of a GUI, which allows to display important metrics during a simulation. On the other hand, the simulator can run without a graphical interface to reduce the consumed memory, which may be utilized for highly scalable simulations.

5.1 Configuring a scenario

5.1.1 Syntax and semantic of the configuring XML file

Within PeerfactSim.KOM, every scenario is defined by a configuration file, to which we refer as *config-file* in the following. The configuration is represented by a XML-file, that can be divided into several formal parts. In order to explain these parts, an example of a typical config-file is given in Listing 5.1, while we subsequently describe the impact of every part on the simulator and denote the possible elements, which a part may embody[1].

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <Configuration>
3    <!-- General Settings -->
4    <Default>
5      <Variable name="seed" value="942" />
6      <Variable name="size" value="252" />
7      <Variable name="end" value="120m" />
8    </Default>
9
10   <!-- Simulator Core -->
11   <SimulatorCore class="de.tud.kom.p2psim.impl.simengine.Simulator"
12     static="getInstance" seed="$seed" finishAt="$end">
13   </SimulatorCore>
14
15   <!-- Components -->
16   <NetLayer class="de.tud.kom.p2psim.impl.network.simple.SimpleNetFactory">
17     <LatencyModel
18       class="de.tud.kom.p2psim.impl.network.simple.SimpleStaticLatencyModel"
19       latency="10" />
20   </NetLayer>
21
22   <TransLayer class="de.tud.kom.p2psim.impl.transport.DefaultTransLayerFactory" />
23
24   <Chord class="de.tud.kom.p2psim.impl.overlay.dht.chord.KBRChordNodeFactory"
```

[1] We assume that the general reader of this documentation is familiar with the basic XML knowledge. For more information on XML (eXtensible Markup Language), we refer to http://www.w3.org/XML/

```
25      port="400" />
26
27    <Monitor class="de.tud.kom.p2psim.impl.common.DefaultMonitor"
28      start="0" stop="$end">
29      <Analyzer class="de.tud.kom.p2psim.impl.analyzer.ChordStructureAnalyzer" />
30    </Monitor>
31
32    <ChurnGenerator class="de.tud.kom.p2psim.impl.churn.DefaultChurnGenerator"
33      start="1m" stop="$end">
34      <ChurnModel class="de.tud.kom.p2psim.impl.churn.ExponentialChurnModel"
35        churnFactor="0.5" meanSessionLength="60m" />
36    </ChurnGenerator>
37
38    <!-- HostBuilder -->
39    <HostBuilder class="de.tud.kom.p2psim.impl.scenario.DefaultHostBuilder"
40      experimentSize="$size">
41      <Host groupID="GlasgowCity">
42        <NetLayer />
43        <TransLayer />
44        <Chord />
45        <Properties enableChurn="false" />
46      </Host>
47
48      <Group size="50" groupID="LatinAmerica">
49        <NetLayer />
50        <TransLayer />
51        <Chord />
52        <Properties enableChurn="false" />
53      </Group>
54    </HostBuilder>
55
56    <!-- Scenario actions -->
57    <Scenario class="de.tud.kom.p2psim.impl.scenario.CSVScenarioFactory"
58      actionsFile="config/actionExample.dat"
          componentClass="de.tud.kom.p2psim.impl.overlay.dht.chord.KBRChordNode" >
59      <ParamParser
60        class="de.tud.kom.p2psim.impl.overlay.dht.chord.OverlayKeyParser" />
61    </Scenario>
62  </Configuration>
```

Listing 5.1: Example of a config-file for a scenario with Chord

Before we start to highlight the different parts, we firstly present the general pattern based on the XML-language, which is used to specify all required components for a simulation. In order to have a concrete example as background while explaining the syntax of the pattern, we are using the XML-element, denoted by `NetLayer` (See line 16-20). As it can be seen, every element, that defines a component within a scenario is a sub-element of the root-element `Configuration`. Depending on the type of the defined component and its impact on the functioning of the simulator, the name for the nested element can be arbitrarily chosen, like for the `NetLayer`-element, or it must be selected from a predefined set of names, that is displayed in Table 5.1. The reason for this naming convention results from the fact, that the simulator requires some basic components for its proper functioning. Therefore, the configuration mechanism searches for XML-elements with listed names, that embody the requested functionality, while later on, the simulator can access the components just by their names. Beside the names, Table 5.1 also sketches the task or component, that the name encloses. A detailed description for the entries can be found past to the table.

Predefined name	Functionality of the contemplated component
`Configuration`	Denotes the root element of the XML-file that embodies the configuration
`Default`	Contains some default values for a simulation
`SimulatorCore`	Highlights the component, that implements and represents the basis for the simulator
`HostBuilder`	Embodies the logic for creating the defined amount of peers
`Scenario`	Comprises the functionality to extract the action out of different types of action files

Table 5.1: Predifened Names for the config-file

After examining the naming conventions for the XML-elements, we take a look at the rest of the pattern, again using the `NetLayer`-element as example. As displayed in Listing 5.1, the name of the XML-element is followed by a set of attributes. From these attributes, one of them is named `class` and contains the fully qualified name of the class, that is responsible for implementing the functionality of the component, represented by the enclosing element (like `SimpleNetFactory`[2] inside the `NetLayer`-element). Depending on the type of the component, an element can contain a further attribute, called `static`, that provides the name of a static method of the implementing class, which is used to instantiate the respective component. While the reference of `class` is mandatory for every element, representing a component of the simulator, the `static`-attribute is only necessary for objects, which apply to the Singleton pattern[3], since their constructors are private and cannot be directly invoked out of the config-file. As `SimpleNetFactory` does not apply to the Singleton pattern and therefore omits the `static`-attribute, we state the example of the `SimulatorCore`-element in line 11-13.

After the process of instantiation, the component needs to be configured. Therefore, the required values are provided by additional attributes as well as by nested elements. The difference between the two alternatives manifests itself in the type of the provided information. While attributes only allow for primitive data types and strings, the nested elements represent complex objects, that may be configured by attributes or nested elements again. Regarding our example, one recognizes, that the `NetLayer`-element does not comprise any attributes for configuration, while it contains the nested `LatencyModel`-element, whose implementing class is `SimpleStaticLatencyModel`[4], which is configured by the attribute `latency="10"`. Regarding the nomenclature for the attributes, responsible for configuration, as well as for the nested elements, we refer to Subsection 5.1.2, where the utilized naming convention is explained.

In the following, we highlight the five parts, a general configuration file consists of. Therefore, we refer again to our example in Listing 5.1, where every new part is introduced with a `<!-- ... -->`-tag.

General settings (line 4-8) The first part is optional and specifies diverse variables, whose values may be required several times within the config-file. By providing the required values instead of the variables, one can omit this part. To create a further variable in the config-file, the following element must be nested inside the `Default`-element, as displayed by Listing 5.2. The new variable can then be accessed via `$variableName` during the rest of the config-file.

```
1 <Variable name="variableName" value="variableValue" />
```

Listing 5.2: Definition of a variable within the config-file

[2] `de.tud.kom.p2psim.impl.network.simple.SimpleNetFactory`
[3] See [10] for details of the Singleton Pattern
[4] `de.tud.kom.p2psim.impl.network.simple.SimpleStaticLatencyModel`

Simulator Core (line 11-13) This part defines the component that implements and represents the basis of the simulator. Therefore, the contemplated element provides the implementing class and passes the required attributes for a proper configuration. In our example `Simulator`[5] is used for implementing the basis of the simulator, while `finishAt` specifies the duration of the simulation and `seed` provides a seed for generating random numbers. Optionally, one can define a so called `statusInterval`-attribute, that specifies the period of time, after which the simulator will printout the currently simulated time.

Components (line 16-36) This part comprises different types of components for a simulation. The selection ranges from the definition of a factory class for a certain layer of a simulation, over the definition of the monitor for data collection during a simulation to the specification of the utilized churn model. Despite the variety of definable components and their usage, one can divide them into two coarse categories, which are characterized in the following.

The first category contains the components, that represent the layers, which are used to compose the hosts of a simulation. In order to create these hosts with their own instances of the chosen layers, every layer must be represented by its corresponding XML-element within the configuration file. Just as described above, this element firstly denotes the implementing class, which is a factory class in case of the layers[6], while it secondly configures the class with the provided attributes and nested elements. The nomenclature of the elements, representing the factory classes for the utilized layers, does not follow any naming scheme. Instead, one can choose any possible name for this element, as long as it is always referenced by this name within the configuration (See the explanation of HostBuilder). But in order to create structured and comprehensive config-files, we recommend to choose appropriate names for the elements to facilitate the comprehension of the content (e.g. the element, that defines the factory for the instances of the network-layer is named `NetLayer`).

In addition to the previous components, there also exists a second type, which is used to add supplementary functionality to the simulator during a simulation. As already mentioned in the beginning of this segment, the functionality may for instance comprise data capturing or the integration of churn. Unlike to the prior components, the instantiation of these objects is not realized by a factory class, but by the component itself. This fact results from the amount of created objects, which is normally reduced to a single one. So like in Listing 5.3, the representing XML-element passes the implementing class of the component, that is afterwards configured by the remaining attributes and nested elements. Additionally as explained during the introduction of the general pattern, the defining element can contain the `static`-attribute, if the respective component applies to a Singleton Pattern.

```
1 <ChurnGenerator class="de.tud.kom.p2psim.impl.churn.DefaultChurnGenerator"
2   start="1m" stop="$end">
3   <ChurnModel class="de.tud.kom.p2psim.impl.churn.ExponentialChurnModel"
4     churnFactor="0.5" meanSessionLength="60m" />
5 </ChurnGenerator>
```

Listing 5.3: Definition of churn within the config-file

HostBuilder (line 39-54) The HostBuilder-part of a config-file consists of a single XML-Element, which is called `HostBuilder` and specifies the class, that is used to create the hosts for a scenario. Besides the class definition, `experimentSize` provides the amount of hosts, that shall be created for the current scenario. The nested elements define the setups for single hosts, identified by a `host`-element, and for groups of hosts, identified by a `group`-element. Their similarities as well as the differences will be clarified in the following.

[5] `de.tud.kom.p2psim.impl.simengine.Simulator`

[6] For more information about the utilized factory method pattern, we refer to [10]

As already mentioned, this part is responsible for composing the specified amount of hosts with the predefined layers. Therefore, `host`- and `group`-elements enclose nested elements, that define the layers, a host or a group of host consists of. It is important to know, that the assembling of the different layers of a host starts with the lowest and ends with highest one, since an upper layer always needs the subjacent layer to register its listener in order to process incoming messages. To recapitulate, we apply the concept of listeners, because there is no direct information exchange on one level between two different hosts (For a detailed explanation of the information flow as well as of the applied communication mechanism we refer to Chapter (TBD)). In case of violating this bottom-up approach, the simulator will throw a `NullPointerException`. Considering our example, we observe the described manner of assembling the layers within the `host`- and the `group`-element (See line 41-46 and 48-53). Besides the assembling of hosts, there is a further nested element called `Properties`, which does not represent a certain layer, but defines additional properties for the single host or the group. The provided attributes inside that element depend on the class, which implements the `HostProperties`-interface. Currently, this is realized by `DefaultHostProperties`, which just relies on the `enableChurn`-attribute, that in turn specifies, if the surrounding host or group is affected by churn.

After examining the similarities of the two XML-elements, we highlight the differences within the configuration, while Subsection 5.1.3 points out the usage and benefit of the different elements during a simulation. Taking a look at the config-file, one can observe that the `size`-attribute states the only difference between the definition of a `host`- or a `group`-element. While the first one always defines one host, the amount of hosts of a `group`-element is set by the named attribute.

To conclude the building process of hosts, we take a look at the `groupID`-attribute, whose relevance and functionality depends on the chosen implementation of the network-layer. In case of complex implementations of the network-layer, like `GnpNetLayer`, valid values of `groupID` are used to retrieve further information from an additional configuration file, called `measured_data.xml`[7]. Available values for `groupID` can be retrieved from the additional configuration file out of the `id`-attribute from the `Host`-elements. `measured_data.xml` offers supplementary information, that is used by the implementing classes to create a more realistic approach as far as the functionality of the simulated network and the exchange of data are concerned. For a detailed explanation of the functioning of `GnpNetLayer`, we refer to Chapter 3.1, while the underlying theoretical model of that implementation is explained in [**?**]. In case of a simplified implementation for the network-layer like `SimpleNetLayer`, `groupID` may contain any string as long as it is unique. The latter condition counts for simple as well as for complex implementations. Besides the just explained usage within the configuration, the `groupID`-attribute is also used to assign actions to a group or to single hosts inside the action file, which will be explained in details within Subsection 5.1.3.

Scenario actions (line 57-61) Within this part, the definition and scheduling of actions for a simulation takes places. This means, that already the surrounding `Scenario`-element in the config-file can contain the actions for the hosts, or the element is used to provide the settings for reading the actions out of an additional action file. Regardless of the used technique, we must provide a class by the `class`-attribute, that implements the `ScenarioFactory`-interface. So, if we want to embed the actions directly into the config-file, Listing 5.4 shows, how the content of `Scenario` with `DOMScenarioFactory`[8] as implementing class could look like.

```
1 <Scenario class="de.tud.kom.p2psim.impl.scenario.DOMScenarioFactory">
2   <Action hostID="host1" time="10">connect</ Action>
3   ...
4 </ Scenario>
```

Listing 5.4: Embedding of actions in the config-file

[7] In case of running simulations with complex implementations for the network-layer, `measured_data.xml` can be downloaded from http://www.kom.tu-darmstadt.de

[8] `de.tud.kom.p2psim.impl.scenario.DOMScenarioFactory`

Although, this is a quite simple and easy approach for the configuration of actions, we focus on the second approach, which uses a separate file for the actions. Therefore, we explain, how the content of `Scenario` must look like in order to retrieve the actions out of the file, while Subsection 5.1.3 outlines its contents and functioning. Using this second approach for the actions, `CSVScenarioFactory`[9] is the implementing class of the `Scenario`-element, while `actionsFile` specifies the location as well as the name of the action file. In order to identify the class, which provides the functionality of the listed action methods for the different components inside the action file, `Scenario` contains a further attribute, called `componentClass`, that denominates the required class. As just one class, offering the implementations for all action methods of different components, is insufficient, the `Scenario` possesses a second attribute, called `additionalClasses`, that can be optionally used to identify further classes, which implement and provide the functionality for the remaining action methods. `additionalClasses` can specify multiple classes, which must be separated by a semicolon. Besides the configuration of the implementing class for `Scenario` by the attributes, one can nest additional elements, e.g. `ParamParser`, that parse the parameters of the action methods and convert them into an adequate format for the methods of the implementing classes. These elements, representing and defining the parsers, are required, if the parameters of the action methods are not just primitive data types, strings or classes but complex objects.

After describing the syntax of the configuring XML-file as well as the semantic of the five identified parts, which in turn consist of several elements, the following subsection focuses on the internal processing of the config-file.

5.1.2 Examination of the configuration process

This subsection focuses on the processing of the config-file within the simulator and describes, which components are responsible for the data extraction. Therefore, we firstly highlight the parsing of the configuration file and subsequently explain the instantiation of the objects, that were defined in the different parts within the config-file. It is important to know, that the following explanation just gives an overview of the contemplated mechanisms, while further details can be found directly in the source code. Therefore, the responsible and implementing classes are referenced, where required.

In order to obtain the information for the configuration of a simulation, the simulator needs the location as well as the name of the config-file at startup. Depending on the chosen procedure of launching, this information must be provided as an argument through the command line or the user can choose from a set of configurations, if he utilizes the graphical user interface (see Figure 5.3a). In the first case, the user may also specify a space-separated list of additional variables, that can be used to override some default values in the config-file. After the provisioning of the needed values during the launching of the simulator, for which a detailed overview is given in Section 5.2, the class, accounting for the configuration, gets the passed information. In the following, the configuring component or class can also be denoted as *configurator*. It is important to know, that a user may write its own configuring component, that provides the functionality for the configuration of a scenario as long as it implements the `Configurator`-interface[10]. Currently, the simulator just offers `DefaultConfigurator`[11] as implementation, which is ready to run simulations and suffices for most of the scenarios. Subsequently to the initialization of `DefaultConfigurator`, the space-separated list of additional variables is parsed and stored by the configurator. As mentioned in the beginning of this subsection, the provided variables are used to override the default values in the config-file. Therefore, these variables must be already defined within the config-file, otherwise they are stored by the configuring component, but never read during the configuration. So for example, if we provide an additional argument like `seed=100`, the config-file

[9] `de.tud.kom.p2psim.impl.scenario.CSVScenarioFactory`
[10] `de.tud.kom.p2psim.api.scenario.Configurator`
[11] `de.tud.kom.p2psim.impl.scenario.DefaultConfigurator`

must contain <Variable name="seed" value="942" /> as nested element within the Default-element in order to access it via $seed during the rest of the file (See Subsection 5.1.1 for a detailed explanation of the config-file and the definition of default values). Having stored the provided values, the simulator registers itself at the configuring component, while afterwards the config-file is finally read. Therefore, DefaultConfigurator utilizes a SaxReader from the dom4j-library, that creates an Document[12]-object, allowing for a random access of the several XML-elements. Due to this type of access, the order of the definition of the different elements can be freely chosen within the config-file, while the configurator may provide the guidelines for the order of processing. Nevertheless, it is helpful to maintain a structure of the config-file as displayed in Listing 5.1, since it facilitates the reading and understanding of the contained information.

After reading the config-file, the created document is traversed, and for every passed element the configurator tries to instantiate the respective object. Figure 5.1 highlights this process within step six to ten, while step one to five displays again the aforementioned procedure of starting the simulator and reading the config-file. In the following, the single steps of the figure, especially step six to ten, are explained.

1. The simulator is started and gets the config-file and an optional space-separated list of additional variables. If it is started with a graphical user interface, the user can choose the config-file from an explorer.
2. The component responsible for the configuration of the scenario is instantiated.
3. The space-separated list of additional variables is processed and stored.
4. The class, representing and implementing the simulator, passes the control flow to the configurator in order to configure the required components.
5. The configuring component parses the passed config-file and creates a XML-tree from the information.
6. Out of this tree, the configurator uses the first level to instantiate all specified objects. Therefore, this step executes a loop, whose amount of iterations depends on the number of elements at the first level. Among these elements, the loop differentiates between the definition of variables, which are surrounded by the Default-element and between the components, representing hosts, additional functionality for the simulator and so on. The first type of elements is examined in the following step, while the latter one is explained afterwards.
7. Within this step, the configurator creates the predefined variables and values out of the config-file. The contemplated step is only executed, if the Default-element at the first level was reached. After setting the variables, the configurator jumps back to the loop of the previous step.
8. Usually, the loop will jump to this step, as the creation of variables only comprises one element, while the rest of elements is related to the instantiation of components for the simulator. Regardless of the handled component, that shall be created, the process of creation is always the same and will be explained in the following, using the NetLayer-element as example.
   After detecting a new XML-element on the first level, the complete element, including its attributes and nested elements, is provided to the createComponent-method of the configurator, which is responsible for the creation of the described object. Within this method, the class-attribute, that contains the fully qualified name of the implementing class, is read. Via Reflection[13] a new instance of this class is created, using the default constructor. If the constructor of the implementing class is not public (e.g. in order to apply to the Singleton pattern), the value of the static-attribute, denoting a static method, is taken to create an instance of the class. In case of the NetLayer-example, the implementing class, provided by class-attribute, is instantiated via its default-constructor, while the static-attribute is omitted. Depending on the class-hierarchy of the instantiated object, supplementary methods can be called within the createComponent-method. For further information

[12] org.dom4j.Document
[13] http://java.sun.com/docs/books/tutorial/reflect/index.html

about these calls of supplementary methods, we refer to the documentation of the `Configurable`[14]- and the `Composable`[15]-interface, while we focus on the further steps of the configuration process.

9. Within this step, the attributes of the instantiated component are parsed and utilized to configure the newly created object. This type of configuration of an object is chosen, as the instantiation is either executed by the default constructor or by a static method (see the previous step), which both do not allow for a direct configuration during the instantiation. Therefore, `configureAttributes(...)` is called to parse the primitive data types for the configuration. So for each attribute `someAttributeName` within the element, the configurator searches for the appropriate method `setSomeAttributeName` within the newly instantiated object of the implementing class. To avoid problems concerning the processing of the attributes, the signature of the *setter*-method should be unique and just provide one single parameter of the respective primitive type. The value of the attribute is converted accordingly to the expected parameter type and the setter-method is invoked with this value as parameter.
   Unfortunately, the `NetLayer`-example does not provide any configuring attributes on the first level for the component, except the mandatory `class`-attribute. But within the next step, we will refer to our example, as it contains a nested element, that also includes configuring attributes.
10. Concerning the configuration of the instantiated components by complex objects, the last step sketches the process of parsing the nested elements of which a component can consist. Due to the fact, that a nested element may also comprise further nested elements, they are processed in a recursive way. Taking the instantiation of a sub-component within one step of the recursion into account, the nested element is instantiated like a normal component, described in the eighth step. So through the `createComponent`-method of the implementing class, specified by the `class`-attribute, the sub-component is created and later on configured by its own attributes as well as nested elements. Considering the configuration of a component by its sub-components, it is similar to the configuration by attributes, so for every nested element `someNestedElementName`, the configurator searches for the appropriate method `setSomeNestedElementName` within the instantiated object of the implementing class.
    Referring to the `NetLayer`-example, the implementing class `SimpleNetFactory` contains the method `setLatencyModel(NetLatencyModel model)`, that is used to set the nested element `LatencyModel`. Taking this element into account, it becomes apparent, that for the `latency`-attribute the implementing class `SimpleStaticLatencyModel`[16] contains the respective setter-method `setLatency(long staticLatency)`.

5.1.3 Syntax and semantic of the action file

This subsection deals with the action file for a scenario and explains the syntax of a valid file comprising the right format of an action command as well as the relation to the config-file. Due to the self-explanatory structure of the file and its commands, this subsections avoids an explanation, that goes into details concerning the source code and the transformation of the commands into valid Java code. Therefore, we refer to the implementing classes, that are denoted in the config-file within the `Scenario`-element, e.g. `CSVScenarioFactory`[17] or `OverlayKeyParser`[18]. For a detailed description of the `Scenario`-element, we refer to Subsection 5.1.1, where the required elements and attributes for a valid configuration are listed and explained. Leaving the topic of setting up the `Scenario`-element and providing the information about the action file, we focus on the semantic and syntax of that file by referring to Listing 5.5, that displays an example of a valid action file. The content of the file is written in

[14] `de.tud.kom.p2psim.api.scenario.Configurable`
[15] `de.tud.kom.p2psim.api.scenario.Composable`
[16] `de.tud.kom.p2psim.impl.network.simple.SimpleStaticLatencyModel`
[17] `de.tud.kom.p2psim.impl.scenario.CSVScenarioFactory`
[18] `de.tud.kom.p2psim.impl.overlay.dht.chord.OverlayKeyParser`

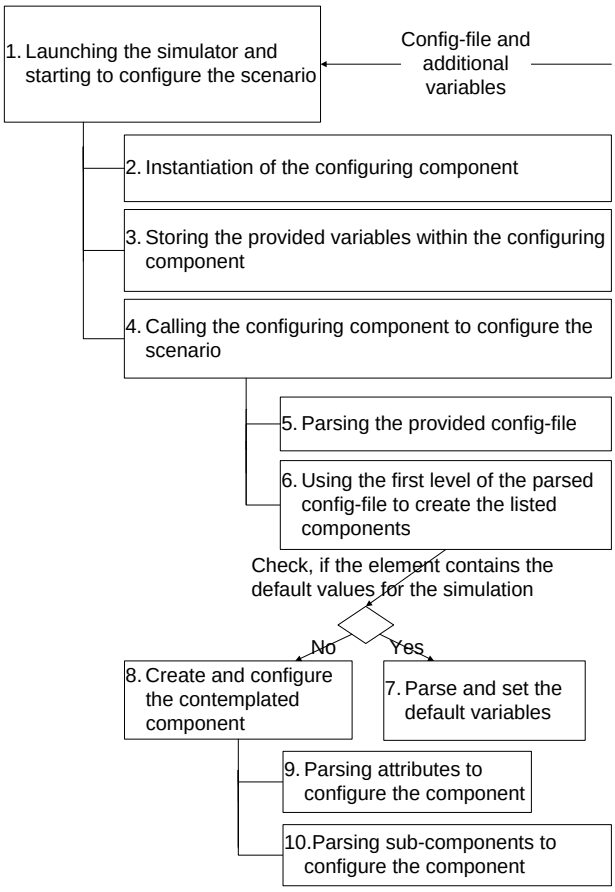


Figure 5.1: Processing of the config-file within the simulator

a pseudo-code-like language and describes what kind of actions a host or a group of hosts must execute and when or in which period the execution shall take place. It consists of comments, which must always be started with a #-character and of action commands, that omit a line delimiter. Instead, every action command must fit exactly to one line, so a command may neither be defined on two lines nor may a line comprise more than one command.

```
1  # Column meanings (format):
2  # Host-Id(String) time(long) opName(String) parameters(list of Strings)
3  # ------------------------------
4  # Scenario with the name chord-actions
5
6  GlasgowCity 1m join callback
7  LatinAmerica 2m-50m join callback
8
9  # Some comments for the action-file
10 GlasgowCity 102m store data1 data1 callback
11
12 LatinAmerica 105m-250m valueLookup data1 callback
```

Listing 5.5: Example of an action file for a scenario with Chord

In the following, the syntax as well as the semantics of a valid action command are clarified and we highlight the terms and conditions for each part of a command in order to demonstrate the possible scopes of the different parts. For this purpose, Listing 5.5 consists of some examples, while the general structure of a command is highlighted in the second line of the file and will be explained in the following.

Host-Id This element specifies the host or the group of hosts for which the defined command shall be executed. Host or group are referenced and summarized by an ID, that must match the value of the `groupID`-attribute of the nested `Host`- or `Group`-element inside the `HostBuilder`-element in the config-file. The example of the config-file contains *GlasgowCity* and *LatinAmerica* as values of `groupID` for the host and the group of hosts respectively. Hence, the `Host-Id`s of the commands within the action file always contain the two IDs.

time This part concretizes the point in time, when the action shall be executed. For the definition of this event, there are two possibilities, which depend on the number of hosts, that `Host-Id` covers. For a single host, the action is just executed at a single point in time, which will be denoted with e.g. `1m`. In case of a group of hosts, one can choose the just mentioned method, that results in an execution of the action from all hosts at the same time. Alternatively, a period of time can be defined, wherein all hosts of that group consecutively execute the action at equidistant distributed points during this interval. A valid definition for that period may look like `2m-50m`. The user can choose from the time units, that are listed in Table 5.2.

opName This element specifies a method of an implementing class, that shall be executed as action for the host(s) at the given time. For the proper functioning, the name of the denoted method must exactly match the signature of the implemented method from the class. To avoid naming collisions and to directly identify the implementing class, the config-file contains the `componentClass`-attribute within the `Scenario`-element, that denotes the respective class. In the provided example of a config- and action file, `componentClass` denotes `KBRChordNode`[19], whose class-hierarchy implements the requested `join`-method.

In order to extend the set of available actions, the `Scenario`-element may comprise the `additionalClasses`-attribute, that specifies a semicolon-separated list of classes, which provide further methods for the action file (We refer to Subsection 5.1.1 for more information about the `Scenario`-element). If a method of a class from the list shall be used within the action file, the respective name of that class must be provided as well. So, if one wants to use a `join`-method from the class `SomeClassName` out of the list, the respective `opName` must look like `SomeClassName:join`. Generally it is important to know, that every method, which shall be executed from the action file, must be declared as `public` within its implementing class.

parameters In order to successfully invoke a method out of the action file, not only the provided name must match the signature of the method within the implementing class, but also the amount of parameters. Therefore, the `parameters`-element just depends on the method, that is specified by `opName`. Parameters of a primitive datatype are ausearchstomatically converted to the required type of the respective argument of the method, while for complex objects, a nested element within `Scenario` (e.g. `ParamParser` within Listing 5.1) specifies a class, that creates the required object out of the passed parameters.

Abbreviation	Time unit
`ms`	Millisecond
`s`	Second
`m`	Minute
`h`	Hour

Table 5.2: Available time units for the action file

[19] `de.tud.kom.p2psim.impl.overlay.dht.chord.KBRChordNode`

5.2 Running a configured scenario

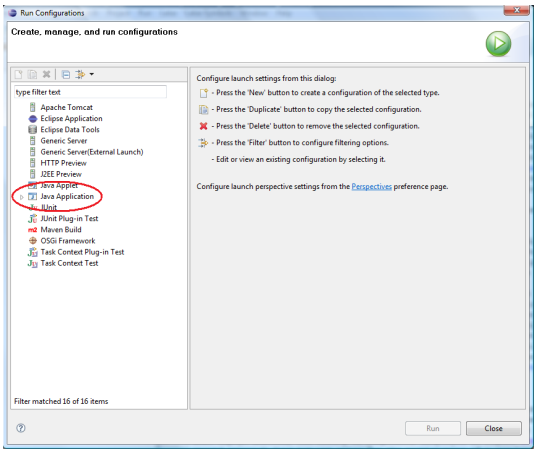
After the description of the required components for a simulation, like config- and action file, this section deals with the launch of a simulation. Therefore, as already sketched in the previous section, we outline which files are required for a successful start and additionally describe the different activation mechanisms of the simulator. Since PeerfactSim.KOM was and is still developed within the integrated development environment *Eclipse*[20], this section only highlights the procedure of starting the simulator out of the IDE, while we omit the launching from the command line. Generally, the provided arguments for the virtual machine and the program do not differ on the command line, but the user is responsible to compile the whole project and to set the classpath correctly. In order to run the simulator on headless systems like servers, we advise to use *ant*[21] for compiling, while the launch of the simulator including all parameters may be executed by a batch-file or a shell-script. Returning to our setup, the rest of this part shows, how a simulation with and without the GUI-component of the simulator is started within Eclipse.

First of all, we concentrate on the startup without the GUI and display the different steps, which are illustrated by Figures 5.2

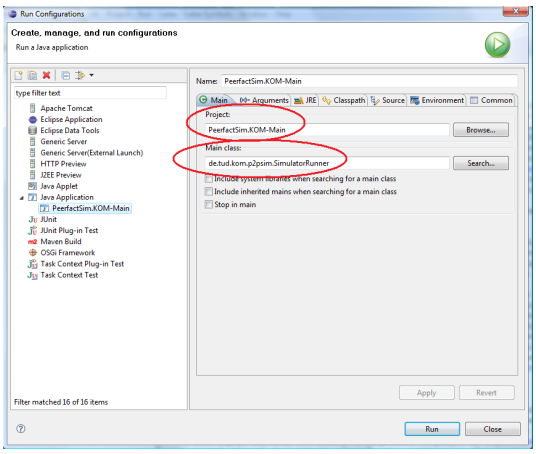
1. Within Eclipse, we are using the *Run Configuration*-dialog to determine the type of configuration for our setup. As shown be Figure 5.2a, the simple *Java application*-configuration is used for PeerfactSim.KOM.
2. As next step the appropriate project out of the workspace is chosen, followed by the delivery of the *Main class*. In case of starting PeerfactSim.KOM without its GUI, `SimulatorRunner`[22] is chosen.
3. The last step for the launching of PeerfactSim.KOM, displayed in Figure 5.2c, comprises the provisioning of the parameters for a simulation. The arguments for the program, listed in the upper field of the dialog, consist of the destination of the config-file including its filename followed by a space-separated list of additional variables. As already indicated in Subsection 5.1.2, this optional list is used to change the default values of the defined variables within the config-file. An valid element of the list looks as follows: `parameterName=parameterValue`. While the space-separated list is optional, the information about the config-file is mandatory.
   Besides the arguments for the simulator, the virtual machine must also be configured, since default setup does mostly not suffice to start a scenario within PeerfactSim.KOM. Therefore, `-Xmx` and `-Xms` are provided to adjust the maximal and minimal amount of memory, that can be consumed by the virtual machine during a simulation[23].

The startup of the simulator with the GUI component is akin to the one explained above. The modifications for this type of launching concern the denomination of the *Main class* within the first step of the setup configuration as well as the provided program arguments in the last step. `GUIRunner`[24] is provided for the *Main class*, while the field for the program arguments is left empty. Instead, the just opened dialog of the simulator, displayed in Figure 5.3a, is used to provide the config-file for a simulation. Therefore, it displays the content of the `config`-directory, which is used by default as directory for the config- and action files. Unfortunately, the contemplated approach does not yet allow for the provisioning of a simulation with additional parameters. This feature is planned to be provided in a later version of the simulator. The main advantage of using the GUI component is that the user can continuously check the status of the simulation. Therefore the window, depicted in Figure 5.3b, periodically measures important data of a simulation and displays the detected values. Furthermore it gives an approximation about the remaining simulation time.

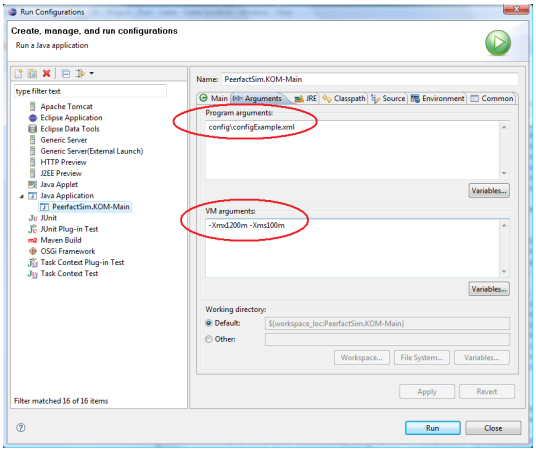
[20] See http://www.eclipse.org for details
[21] See http://ant.apache.org for details
[22] `de.tud.kom.p2psim.SimulatorRunner`
[23] For a detailed description and influence of the listed VM-parameters, we refer to http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp
[24] `de.tud.kom.p2psim.GUIRunner`



(a) The *Run Configuration*-dialog of Eclipse

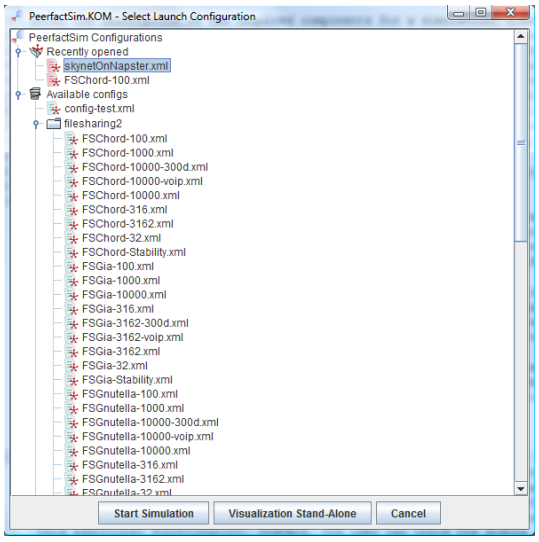


(b) The dialog for the general project specifications

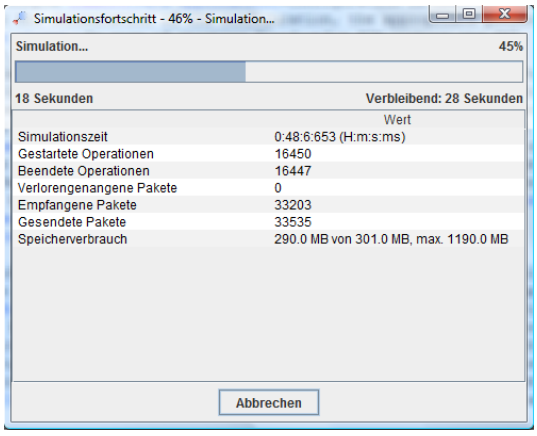


(c) The dialog for the parameter settings

Figure 5.2: Different configuration steps for the launching of PeerfactSim.KOM



(a) GUI component of PeerfactSim.KOM for choosing config-files



(b) Status window of PeerfactSim.KOM monitoring a simulation

Figure 5.3: Different GUIs of PeerfactSim.KOM

6 Visualization

This chapter deals with the visualization component of the simulator. The component is an add-on to the console based simulator and was mainly developed to give people new to the simulator a good understanding of what the simulation of P2P-networks is about. It may also be helpful during the development of new components, especially P2P-overlays, or to measure performance metrics (for small- and medium-sized network scenarios). It is possible to easily extend the visualization component and customize it for special purposes.
To visualize a scenario the component is registered as a monitor for the simulation (See chapter 4 for general information about the monitoring and analyzing architecture of PeerfactSim.KOM). As monitor the component gathers all information it later needs for a visualization. After the simulation process is finished the visualization window opens and allows to visualize the scenario based on the gathered data. The usage of the graphical user interface is aimed to be intuitive as the basic functions are quite similar to common video players. It is possible to save the data of a whole simulations and load it later on when a visualization is needed.

Even if it should be clear after the last paragraph we want to clarify again that the visualization does not take place in real-time during a simulation but acts as recorder that allows a visualization after the simulation.

6.1 Limitations

The visualization component aims to visualize small to medium scenarios. We do not suggest to use it for scenarios that contain a huge number of hosts or produce a lot of messages. The reason is that for big scenarios the images generated for visualization tend to get unreadable. A lot of overlapping shapes and labels are drawn which makes the pictures useless. An other problem with huge scenarios is the limitation of the rendering speed. Since the visualization keeps track of a lot of details it has to work with complex data structures. These structures are traversed during the visualization and are used to render an image. Testing results show that an average scenario greater than 80 nodes will result in a maximum image rendering rate less than 1 image per second on a common workstation (Centrino Duo, 2GB RAM). Therefore we do not recommend the usage of this component for such large scenarios.

6.2 The structure of the visualization component

The following section describes the visualization's structure in an abstract way. The basic structure should become clear and allow you to extend the component to your needs. As the component consists of several packages and classes we only pick out the main concepts. First we describe the part responsible for the link to the simulation core. This part enables the gathering of data that later on is used for the visualization. After that we describe the main architecture of the actual visualization component and the internally used data structures.
To use the visualization for a already included overlay or to visualize a new overlay chapter 6.2.1 should give you enough information to do so. Chapter 6.2.2 gives you further insight and may help you to extend the visualization with new features.

6.2.1 Information gathering

To describe the main structure of the information gathering we want to give an overview about the important classes that exist for that purpose. Figure 6.1 shows these classes and a subset of the methods

they define. We left out some details as they distract from the main concept. Please take a look at the source code itself and explore the available methods if you plan to extend the classes. The central class

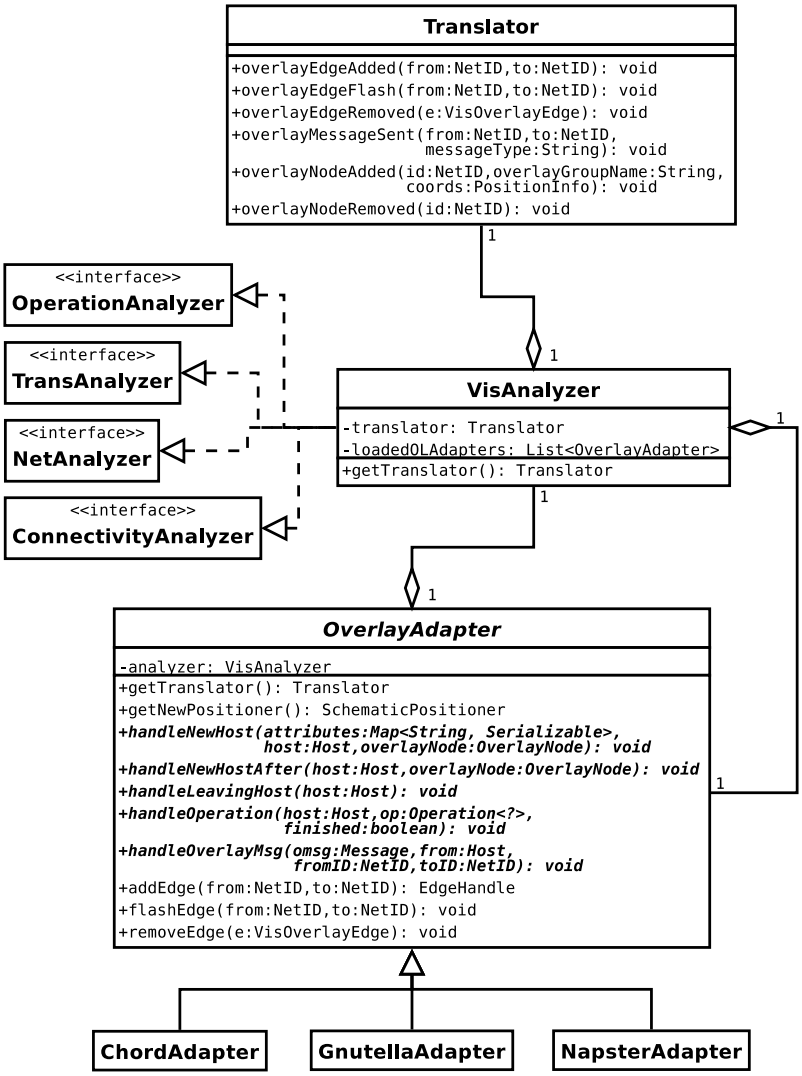


Figure 6.1: Classes responsible for information gathering

in this constellation is `VisAnalyzer`[1]. This class implements the analyzer interfaces (see Chapter 4 for more details on this interfaces) and so enables the gathering of simulation data by using well defined connection points to the simulator.
The class `OverlayAdapter`[2] is tightly coupled to the `VisAnalyzer` and defines an abstract base class for specific visualization adapters. Such a specific adapter encapsulates the behavior of the whole visualization component as it defines what simulations events are tracked for the later visualization and what changes they include for the generated images. There already exist some adapters such as `ChordAdapter`[3] or `GnutellaAdapter`[4]. They define a behavior for useful visualizations of specific overlay implementations.

[1] `de.tud.kom.p2psim.impl.util.vis.analyzer.VisAnalyzer`
[2] `de.tud.kom.p2psim.impl.util.vis.analyzer.OverlayAdapter`
[3] `de.tud.kom.p2psim.impl.util.vis.analyzer.ol.chord.ChordAdapter`
[4] `de.tud.kom.p2psim.impl.util.vis.analyzer.ol.gnutella.GnutellaAdapter`

From a more abstract point of view an overlay adapter collects overlay-specific visualization information out of the runtime behavior of an overlay implementation. This process includes a set of events that can be handled within the adapters to generate useful visual representations of the simulations. Here is a list of events that the adapter is informed about and the developer may handle within its implementation of the class (The according methods are given in brackets):

1. The joining of a host (`handleNewHost`, `handleNewHostAfter`)
2. The leaving of a host (`handleLeavingHost`)
3. The initiation of an operation (`handleOperation`)
4. The complete transmission of an overlay message (`handleOverlayMsg`)

The called methods are defined as abstract within the class `OverlayAdapter` and are implemented when writing a new specific adapter. For reference implementations have a look at already existing adapters in the appropriate package[5].

To actually visualize caught events the classes `OverlayAdapter` and `Translator`[6] define several methods that you can use to generate changes within the visualization. All of the method calls result in a visualization event that internally is put into a data structure that we call the event time-line. This structure groups all triggered events by the time they were generated. Later on, during the visualization process the event time-line is traversed and the events are visualized. The time-line and the internal architecture are described in chapter 6.2.2.

The following visualization events are possible (The according method names are given in brackets. There may be several methods with the same name but different parameter combinations - take a look at them to pick the one that helps you best):

1. Add a new overlay node (`overlayNodeAdded`)
2. Remove an overlay node (`overlayNodeRemoved`)
3. Add a new edge between two nodes (`overlayEdgeAdded`)
4. Remove an edge between two nodes (`overlayEdgeRemoved`)
5. Add an edge that removes itself after a short time interval (`overlayEdgeFlash`)
6. Indicate the sending of a message between two nodes (`overlayMessageSent`)
7. Indicate the change of a attribute assigned to a node (`nodeAttributeChanged`)

To keep the amount of duplicated source code minimal the class `OverlayAdapter` defines some extra methods that implicitly add some basic attributes to an event. At the moment this just pertains to the generation of edges. The following method are available:

1. `addEdge` - The name of the used overlay is automatically added to the attributes and a type name for the edge may be provided as String.
2. `flashEdge` - The description of addEdge also holds here.

Hint: You may have noticed that the methods to add edges return an object of type `EdgeHandle`[7]. You can use this object to ease a later removal of that edge. Therefore keep a reference to that object and later on simply invoking the method `remove` on it.

[5] `de.tud.kom.p2psim.impl.util.vis.analyzer.ol`
[6] `de.tud.kom.p2psim.impl.util.vis.analyzer.Translator`
[7] `de.tud.kom.p2psim.impl.util.vis.analyzer.Translator.EdgeHandle`

6.2.1.1 The positioning of hosts

Since the simulator is very modular and allows to freely replace implementations of layers or components it also supports different positioning schemes for hosts. To enable the interchangeability they all apply to the interface `NetPosition`[8]. The visualization component has to take the different implementations into account to transform positions of hosts used in a scenario to meaningful pixel coordinates on the window they are visualized at.
To encapsulate the logic of this transformation we defined the Interface `INetPositionTransformer`[9]. It allows to define classes that define the transformation of a certain implementation of `NetPosition` to a position used during the visualization. In most of the cases this transformation is only a conversion between types but it could also be used to define complex positioning schemes. We provide simple transformers for existing implementations of `NetPosition`. You can find them in the package `netPosTransformers`[10].

Hint: If you write new transformers, you have to extend the method `transformPosition` in the class `VisAnalyzer` to allow the visualization to use them.

There is also another concept concerned with the positioning of hosts that tries to generate positions not from the `NetPosition` of hosts but also from other properties such as the overlay id of a host. This is used to provide schematic positioners that the visualization uses as alternative to the `NetPosition` driven approach. For that purpose the class `OverlayAdapter` provides a method to retrieve a so called `SchematicPositioner`[11] (`getNewPositioner`). By default a ring based positioning is used if the special `OverlayAdapter` does not override the method.

6.2.2 Internal architecture

This subsection is meant to give an overview about the internals of the visualization component. It does not take into account the gathering of information from the simulator as that is described in chapter 6.2.1.
Figure 6.2 shows the classes responsible for the internal representation of the overlay graph that is visualized and its processing through visualization events. The core component of this part is the class `VisDataModel`[12]. It maintains references to instances of the classes `VisOverlayGraph`[13] and `EventTimeline`[14]. `VisOverlayGraph` contains the description of the overlay graph with the currently existing nodes and edges, while `EventTimeline` organizes the events that manipulate the overlay graph at a certain simulation time. The two classes and their important dependencies are described in the following paragraphs.

6.2.2.1 The overlay graph

As already stated, the class `VisOverlayGraph` as shown in figure 6.2 contains the description of the overlay graph. This description is used to visualize the scenario. Since figure 6.2 is thought as an overview it is simplified in some points. Figure 6.3 shows the class hierarchies and dependencies for the internal graph representation in detail.

[8] `de.tud.kom.p2psim.api.network.NetPosition`
[9] `de.tud.kom.p2psim.impl.util.vis.analyzer.netPosTransformers.INetPositionTransformer`
[10] `de.tud.kom.p2psim.impl.util.vis.analyzer.netPosTransformers`
[11] `de.tud.kom.p2psim.impl.util.vis.analyzer.positioners.SchematicPositioner`
[12] `de.tud.kom.p2psim.impl.util.vis.model.VisDataModel`
[13] `de.tud.kom.p2psim.impl.util.vis.model.overlay.VisOverlayGraph`
[14] `de.tud.kom.p2psim.impl.util.vis.model.EventTimeline`

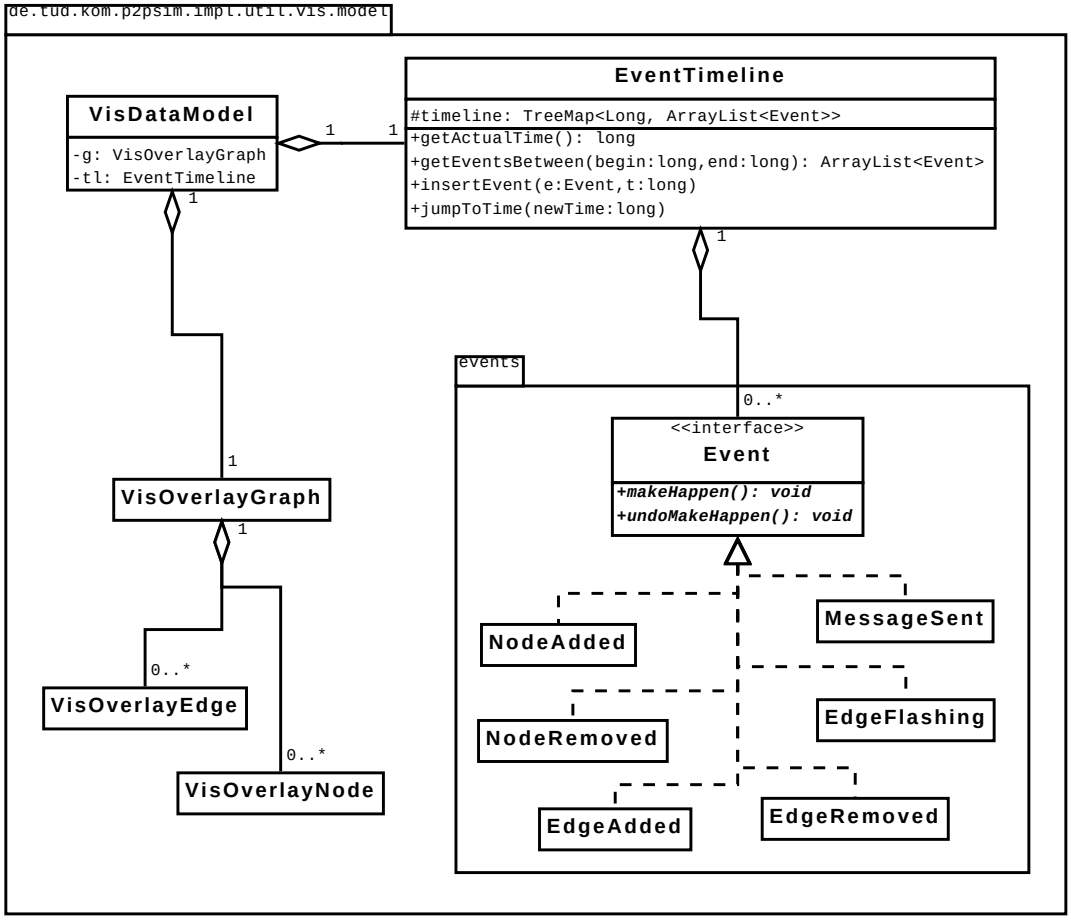


Figure 6.2: Important internal classes

The class `VisualGraph`[15] builds the base class for `VisOverlayGraph` and the classes `Node`[16] and `Edge`[17] for `VisOverlayNode`[18] and `VisOverlayEdge`[19] respectively.
The reason for this structure and the existence of the extra base classes was the idea to make the visualization component interchangeable. At the beginning of the visualization development it was not clear whether the now existing visualization would be the only one or if for example more complex components with support for 3-dimensional visualizations would be added later on. In this case additional extensions to the base classes would become necessary.

To enable the traversal of the graph representation the base class `VisualGraph` provides methods to iterate over members of the graph (`iterate`, `iterateNodes`, `iterateEdges`). To call one of the methods you have to provide an object that complies to the interface `ModelIterator`[20]. Figure 6.4 shows the interface and three implementations. One of it is for example the class `Painter`[21] that traverses the graph to generate a picture of it.

[15] de.tud.kom.p2psim.impl.util.vis.util.visualgraph.VisualGraph
[16] de.tud.kom.p2psim.impl.util.vis.util.visualgraph.Node
[17] de.tud.kom.p2psim.impl.util.vis.util.visualgraph.Edge
[18] de.tud.kom.p2psim.impl.util.vis.model.overlay.VisOverlayNode
[19] de.tud.kom.p2psim.impl.util.vis.model.overlay.VisOverlayEdge
[20] de.tud.kom.p2psim.impl.util.vis.model.ModelIterator
[21] de.tud.kom.p2psim.impl.util.vis.visualization2d.Painter

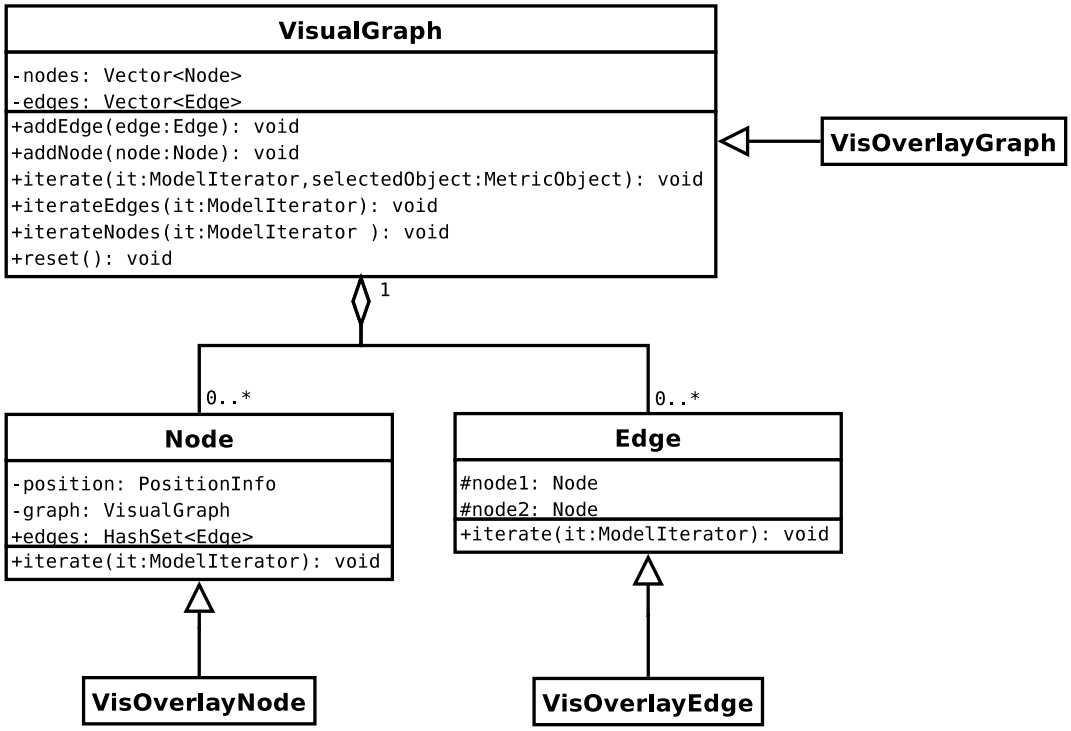


Figure 6.3: The internal overlay graph representation

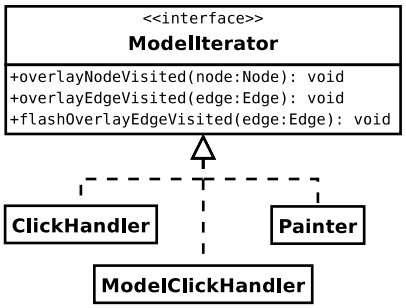


Figure 6.4: The graph iterator classes

6.2.2.2 The event time-line

As seen in figure 6.2 the class `EventTimeLine` is responsible for the organization of visualization events. Events are implementations of the interface `Event`[22] that defines two methods, one to make an event happen and one to undo the event. These two methods normally access directly the overlay graph and change its topology by adding or removing nodes and edges for example. It is important to implement the undo method probably as it is used when the user wants to jump from one point in an visualization to an earlier point. In this case all events that took place since the target time have to be revised. If this action is not possible in a consistent way, it may lead to an inconsistent representation of the whole scenario.
To allow an easy organization of a huge number of events, the class `EventTimeLine` stores the events

[22] `de.tud.kom.p2psim.impl.util.vis.model.events.Event`

in a `TreeMap`[23] with the time of the occurrence as key and an array of events as value. The `TreeMap` automatically maintains the order of the events and allows an easy traversal and extraction of events that lie in a certain time interval.

6.2.3 The user interface

Classes that deal with the user interface can be found within the package `ui.common`[24]. Figure 6.5 shows an overview of important classes that build the user interface.

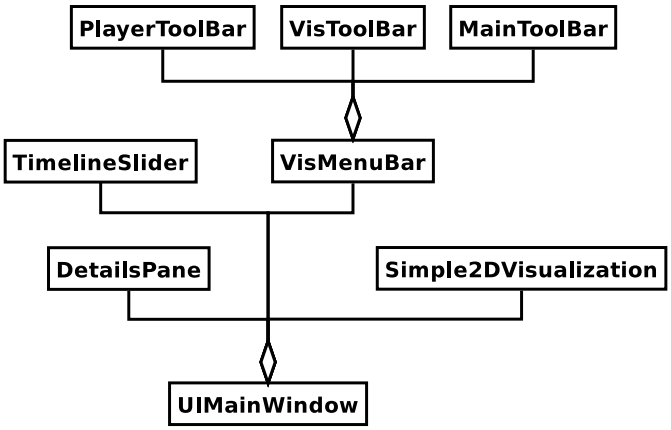


Figure 6.5: Important user interface classes

The class `UIMainWindow`[25] defines the main visualization window. All other user interface components are either contained within this class or created by it. `Simple2DVisualization`[26] defines the main component of the visualization window at which the overlay graph is painted. `DetailsPane`[27] is a small sub-component that displays details about the visualized scenario or a selected entity within the graph. `TimelineSlider`[28] is the component for displaying the slider at the bottom of the window that shows the progress of the visualization and allows to jump from one point in the simulation to another. `VisMenuBar`[29] is responsible for the menus that are shown at the top of the window and contains all sub-menus such as the tool bar for buttons to control the visualization player (`PlayerToolBar`[30]) or the main menu to save/load scenarios and close the visualization (`MainToolBar`[31]). Figure 6.6 shows a screenshot of the main window to get a better idea of the components we described in this chapter.

23 `java.util.TreeMap`
24 `de.tud.kom.p2psim.impl.util.vis.ui.common`
25 `de.tud.kom.p2psim.impl.util.vis.ui.common.UIMainWindow`
26 `de.tud.kom.p2psim.impl.util.vis.visualization2d.Simple2DVisualization`
27 `de.tud.kom.p2psim.impl.util.vis.ui.common.DetailsPane.DetailsPane`
28 `de.tud.kom.p2psim.impl.util.vis.ui.common.TimelineSlider`
29 `de.tud.kom.p2psim.impl.util.vis.ui.common.menubar.VisMenuBar`
30 `de.tud.kom.p2psim.impl.util.vis.ui.common.toolbar.PlayerToolBar`
31 `de.tud.kom.p2psim.impl.util.vis.ui.common.toolbar.MainToolBar`

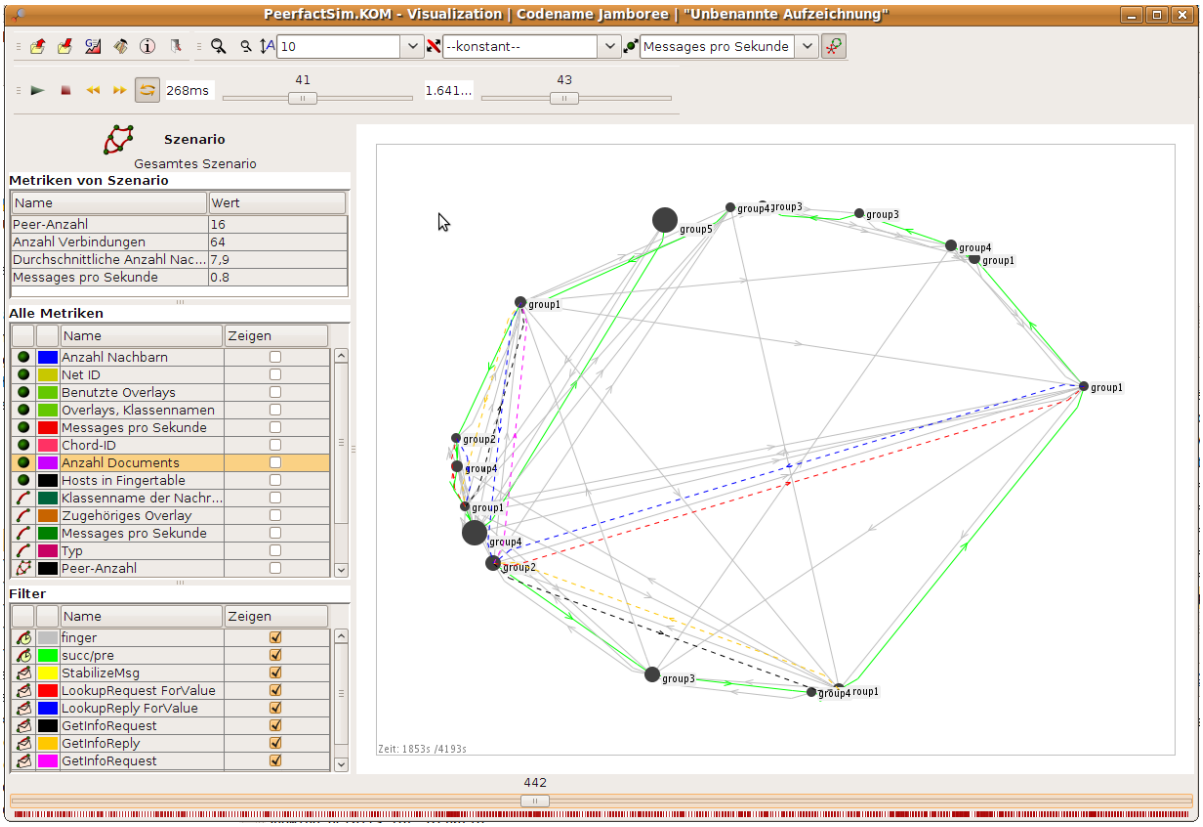


Figure 6.6: Screenshot of the visualization's main window

Bibliography

[1] gnuplot, . URL http://www.gnuplot.info.

[2] The Annotated Gnutella Protocol Specification v0.4, . http://rfc-gnutella.sourceforge.net/developer/stable/index.html.

[3] Information processing systems - Open System Interconnection. International Standard ISO/IEC 7498, ISO, 1989.

[4] FIPS 180-1. Secure hash standard. *U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, VA*, April 1995.

[5] Franz Aurenhammer. Voronoi Diagrams — A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys*, 23:345–405, 1991.

[6] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P Systems Scalable. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407–418, 2003. ISBN 1581137354. doi: 10.1145/863997.864000.

[7] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems*, pages 33–44, 2003.

[8] György Dán and Niklas Carlsson. Power-law Revisited : A Large Scale Measurement Study of P2P Content Popularity. In *Proceedings of the 9th Iternational Conference on Peer-to-peer Systems*, pages 1–11. USENIX Association, 2010.

[9] Paul Francis. Yoid: Extending the Multicast Internet Architecture. In *White Paper*, 2007.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addisson-Wesley, Toronto, Ontario. Canada, 1995.

[11] Prasanna Ganesan, Krishna Gummadi, and Hector Garcia-Molina. Canon in g major: Designing dhts with hierarchical structure. Technical Report 2003-74, Stanford InfoLab, 2003. URL http://ilpubs.stanford.edu:8090/626/.

[12] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 314–329, 2003.

[13] Shun-Yun Hu, Jui-fa Chen, and Tsu-Han Chen. VON: A Scalable Peer-to-Peer Network for Virtual Environments. *IEEE Network*, 20:22–31, 2006.

[14] Shun-Yun Hu, Chuan Wu, Eliya Buyukkaya, Chien-Hao Chien, Tzu-Hao Lin, Maha Abdallah, and Jehn-Ruey Jiang. VAST: A Spatial Publish Subscribe Overlay for Massively Multiuser Virtual Environments. Technical report, VAST, 2010.

[15] Sebastian Kaune, Konstantin Pussep, Christof Leng, Aleksandra Kovacevic, Gareth Tyson, and Ralf Steinmetz. Modelling the Internet Delay Space Based on Geographical Locations. In *Proc. of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 301–310, 2009.

[16] Tor Klingberg and Raphael Manfredi. Gnutella 0.6, 2002. http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html.

[17] Gerald Klunker. A Measurement-Based Approach for Realistic and Efficient Modeling of Transmission Time in Large Scale Peer-to-Peer Simulations. Master's thesis, Technische Universität Darmstadt, 2008.

[18] Aleksandra Kovacevic, Sebastian Kaune, Patrick Mukherjee, Nicolas Liebau, and Ralf Steinmetz. Benchmarking Platform for Peer-to-Peer Systems. *it - Information Technology (Methods and Applications of Informatics and Information Technology)*, 49(5):312–319, Sep 2007.

[19] Aleksandra Kovacevic, Kalman Graffi, Sebastian Kaune, Christof Leng, and Ralf Steinmetz. Towards benchmarking of structured peer-to-peer overlays for network virtual environments. In *Proceedings of the 14th International Conference on Parallel and Distributed Systems*, pages 799–804, 2008.

[20] Gerald Kunzmann, Andreas Binzenhofer, and Robert Henjes. Analyzing and Modifying Chord's Stabilization Algorithm to Handle High Churn Rates. *Information Technology*, pages 376–382, 2005.

[21] Gerald Kunzmann, Robert Nagel, Tobias Hossfeld, Andreas Binzenhofer, and Kolja Eger. Efficient Simulation of Large-Scale P2P Networks: Modeling Network Transmission Times. In *15th Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 475–481, 2007.

[22] Lauinger, Tobias. HKademlia: Routing in a Virtual Hierachy, 2007.

[23] Jaime Lloret, Juan R. Diaz, Jose M. Jiménez, and Manuel Esteve. The Popularity Parameter in Unstructured P2P File Sharing Networks. In *Proceedings of the 4th WSEAS International Conference on Applied Informatics and Communications*, pages 1–7, 2004.

[24] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, pages 53–65, 2002.

[25] Stephen Naicken, Barnaby Livingston, Anirban Basu, Sethalat Rodhetbhai, Ian Wakeman, and Dan Chalmers. The State of Peer-to-Peer Simulators and Simulations. *SIGCOMM Computer Communication Review*, 37:95–98, 2007.

[26] John A. Nelder and Roger Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308, 1965. ISSN 0010-4620.

[27] T. S. Eugene Ng and Hui Zhang. Towards global network positioning. In *Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 25–29, 2001.

[28] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, 2001.

[29] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *ACM Special Interest Group on Data Communication*, pages 149 – 160, 2001.

[30] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002. ISBN 0130661023.