Technische Universität Darmstadt
Fachbereich Elektrotechnik und Informationstechnik
Fachbereich Informatik (Zweitmitgliedschaft)
Fachgebiet Multimedia Kommunikation
Prof. Dr.-Ing. Ralf Steinmetz

How to create an Overlay in PeerfactSim.KOM

Eser Esen

February 28, 2008

Contents

1 Introduction **5**
- 1.1 Objective . . . 5
- 1.2 Requirements . . . 5
- 1.3 How to use this Tutorial? . . . 5

2 Basics **7**
- 2.1 What is an overlay? . . . 7
- 2.2 What and Why PeerfactSim.KOM? . . . 8

3 Create an overlay for PeerfactSim.KOM **9**
- 3.1 Preparations . . . 9
- 3.2 Definitions . . . 9
- 3.3 The basic structure of an Overlay . . . 10
- 3.4 Implementation of OverlayID and OverlayContact . . . 12
- 3.5 The Routing Table . . . 16
- 3.6 Messages . . . 18
- 3.7 The Node Implementation . . . 20
- 3.8 The Operations . . . 22
  - 3.8.1 Operation callbacks . . . 23
- 3.9 The Bootstrap Part . . . 24
- 3.10 Applications . . . 25
- 3.11 Component factories . . . 26
- 3.12 The Routing Mechanism . . . 28
- 3.13 Creating networks with my overlay . . . 29
- 3.14 Evaluation of the new Overlay . . . 32

4 FAQ **35**
- 4.1 Common questions . . . 35
- 4.2 Implementation . . . 35

Chapter 1

Introduction

This paper is intended to show you how to create an overlay in PeerfactSim.KOM. It additionally has a FAQ with questions about the implementation process and answers to some common questions. This tutorial help to understand the basic structure of an overlay designed for PeerfactSim.KOM and give a help for further projects with PeerfactSim.KOM.

1.1 Objective

The objective of this paper is to show the first steps to prepare and create a full functioning overlay with PeerfactSim.KOM. After that, you will be able to create networks simulated with PeerfactSim.KOM to evaluate your overlay.

1.2 Requirements

The first step before we start is to prepare your workspace for creating overlays. Apperantly we need a good environment to implement our overlay. The standard environment we can use is Eclipse in an actual stable release (SDK 3.3.1 or current Europa Edition). The second is to get a copy of PeerfactSim.KOM (currently v3.0). And certainly we need a specification for our new overlay which we want to create now. Or in other words, you should know what kind of overlay you want to create. So we need to ask ourself, how the topology will be built, how the message transmission works (routing messages), how we are going to maintain our peers states.

1.3 How to use this Tutorial?

This tutorial is structured in a top-down manner. It is also possible to use it as a manual to get answers for implementation purposes. The FAQ part in this paper answers some frequent question in common and in implementation.

Chapter 2

Basics

First we start with some basics. We need to know what an overlay is and what precisely PeerfactSim.KOM is. In the next part we will present the design of PeerfactSim.KOM to show how it works and how every overlay, developed for PeerfactSim.KOM, is working with this.

2.1 What is an overlay?

An overlay is a network which is built on top of an existing network. For example: A network built on top of the Internet with some peers connected to each other is called an *Overlay Network*. Hosts in the underlaying network acting in that overlay network are called nodes. These nodes are connected by virtually or logically defined links, which is totally independent from their connection in the underlaying network. A basic example of an overlay could be GNutella [4] which is built on top of the existing Internet with connected hosts.

The main characteristic of an overlay network is defined as follows:

- network built on top of an existing other network
- in most cases an own virtually or logically address space is defined
- uses a routing technique to send messages to other peers in that overlay-network
- peer-to-peer networks are overlay networks (like Skype, Chatnetworks, GNutella, BitTorrent)

An overlay defines with its structure a new topology on top of the underlaying network. It has its own rules and routing technique using the links in the underlaying layers of the existing network.

2.2 What and Why PeerfactSim.KOM?

PeerfactSim.KOM [3] is a java based Peer-to-Peer evualation platform which gives us the ability to create an overlay and simulate large-scale networks with it. The target is to evaluate those overlays and compare each other to get a clear conclusion about the functionality and answers to the most important questions like the efficiency, scalability and flexibility.

Chapter 3

Create an overlay for PeerfactSim.KOM

This chapter describes the implementation process of an overly for PeerfactSim.KOM. First we start with the preparations and the basic structure of an overlay used in PeerfactSim.KOM. In the next part we implement all needed components for an overlay step-by-step. The last part describes the important routing core of an overlay and introduce the using of analyzers for the evaluation process. In this paper, all components are prefixed with *My* to show examplary components.

3.1 Preparations

Before we start with the implementation, we need a concrete specification for an overlay. This specification describes the structure of the concrete components of an overlay. The best way to clarify this, is some type of UML-Diagram or any other model of the overlay we want to create. Based on that model, we start to implement our new overlay. Some important components we have to specify, are the node, the routing mechanism, maintenance processes like repairing the current state (Routing Table), handling failure nodes, doing pings to keep the routing table up-to-date.

One important part is the consistency we have to pay attention for. We're going to create an overlay running on a simulator. Since all of our simulations run on one PC, we need to clone objects when moving them between nodes and other components. If we want to save state information in an exchangable object, then we need to clone it before exchanging it between nodes to keep the consistency in our simulation.

3.2 Definitions

Before we start, we're going to explain some definitions used by this tutorial.

- **Node** is the peer acting on the overlay. The Node is the main part of our implementation. It will handle all message transmissions and many other operations. After implementing all sub-components we're going to bring them all together on our Node.

On a real network a node can be related to many applications, but applications will always have a single relation to one node. So one Node, also called, OverlayNode, can handle many applications message transmissions, while one application is only related to one node.

- **IP (Internet-Protocol)** is one Node's current address on the Internet or Local Area Network (LAN), which is used to uniquely identify the node. In most cases the IP address can be used by the overlay to identify the nodes. In every case, the nodes on top of the hosts connected each other in the underlying network, the IP is used for connection.

3.3 The basic structure of an Overlay

Every overlay consists of different types of components we have to implement. For a properly working overlay, we have to clarify all needed components needed by our overlay.

These are as follows:

- **Identify the Node:** Every node in an overlay network must be uniquely identifiable. This means a node must have some kind of virtual or logical label. Since we identify hosts on the underlaying internet by using their IP address, we have to define or use an existing address space to identify each node in the overlay network. In most cases, overlay networks have their own address space. These addresses are typically virtual or logical labels, which are calculated or randomly chosen. An example is to calculate a hash code using the current IP address. This means we have to decide how to save this kind of virtual or logical label. PeerfactSim.KOM has all needed interfaces and abstract classed to create our overlay components. The component used for identification in PeerfactSim.KOM is called *OverlayID*. Every node in an overlay network is identified by one of these OverlaysIDs. Every time we want to give an object or component a unique identification to identify it on the overlay, we must assign it with an OverlayID. This means all nodes, document objects and messages, which must be identified on the overlay network, have to be identified by the OverlayID interface or at least have a reference to an OverlayID object. For example, messages do not need to be labeled by an OverlayID. Since a message has its destination, this destination is marked by an OverlayID object owned by the current message.

- **One nodes contact card:** If one node wants to send some messages to another node, he needs to get his contact information like the business card of a business man. So we need a component which holds the ID of a node and his contact address. By default, the ID is, like mentioned before, the OverlayID and his address, this means, his IP address. These contacts are called *OverlayContact*. The Interface *OverlayContact* represents one node's 'business card'. With an *OverlayContact* one node is able to send messages or documents to the node represented by this *OverlayContact*. So all

node's are going to hold a list of different OverlayContact's representing a group of nodes (representing a part of the whole network) known by the current node.

- **Identification of messages and data objects:** Every messages and every data object sent from one node to another has its destination. These destinations are specified by OverlayIDs. But like in Hash tables using keys for the objects, which are used to identify and correctly place these objects in the table, we differentiate the ID and the key. So we use *OverlayID* generally to identify any object, but to identify messages and data objects we use *OverlayKey*. OverlayKeys are extending OverlayID by default, this means OverlayKey is from type OverlayID but we need to differentiate the identification of nodes and messages/objects since nodes are acting components with running operations and messages/objects are just objects used and handled by the nodes.

- **The Routing table:** Peers (Nodes) sending messages and data objects, need an address book holding contacts to known peers in the current active network. This address book, called the routing table has a protocol-specific structure. In some cases, peer-to-peer protocols could use 1-dimensional arrays of OverlayContacts, holding one nodes ID and address. Other protocols use complex tree structures to store those OverlayContacts. So, dependent from the protocols specification, we are going to implement our routing table. These kinds of routing tables are represented by the interface *OverlayRoutingTable*. This interface provides the basic methods to access and manage the routing table.

- **Encapsulate commands in operations:** All commands like join, leave, maintenance processes and many others, should be encapsulated in classes according to the command design-pattern [2]. The command pattern encapsulates a request as an object with all its parameters. The interface we have to use for this commands is called *AbstractOperation*, which have to be parameterized. The parameters concretize the node type and the result type returned after finishing the operation. Typically, if some action should take place in a component, this action will be represented by an operation object providing the required functionality. The *AbstractOperation* is an abstract which holds all basic methods ready to use. All of our operations we are going to implement and extend *AbstractOperation* have their routines and algorithms executed while a simulation is running. In the following subchapters we will concretize the implementation of these operations.

- **Creating the components for simulation:** To simulate the network working with our newly created overlay, we have to give the simulator the ability to create the correct components like nodes and applications. The simulator uses an interface for creating components according to the *Abstract Factory* Design-Pattern. So every component that needs to be created by the simulator to run those simulations, must have a factory class implementing that interface, which is called *ComponentFactory*. Typically we have to create factories for creating nodes, IDs and applications running

on top of nodes. The *ComponentFactory* interface represents the class ability to create the needed components for our simulations. In Component factories we are going to feed the components with all needed data and parameters returning them to the simulator by the given method.

The next important part is the concrete topology of the overlay network. In most cases ID spaces are represented by a ring. The following figure shows the simple network in a ring with its IDs in space. The IDs are represented as integer numbers.

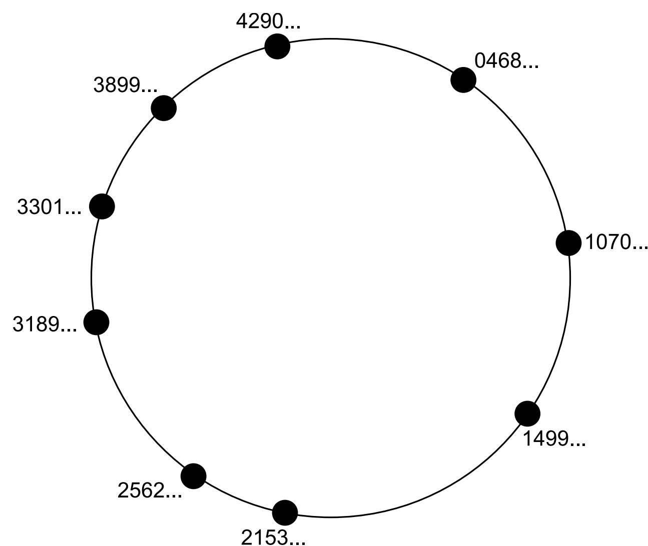

Figure 3.1: A simple ring topology represented by integer ID's

3.4 Implementation of OverlayID and OverlayContact

Now let's start creating the first components we need to create our node component. We use the interface *OverlayID* located in the API Packets of PeerfactSim.KOM. Let us call our ID component MyOverlayID. In the following listing we see an auto-generated class in Eclipse implementing the *OverlayID*-Interface.

Listing 3.1: Example: Automatically generated class MyOverlayID in Eclipse

```
1 public class MyOverlayID implements OverlayID {
2
3         public MyOverlayID() {
4                 // TODO Auto-generated constructor stub
5         }
6
7         public byte[] getBytes() {
8                 // TODO Auto-generated method stub
9                 return null;
10        }
11
```

```
12          public Object getUniqueValue() {
13                  // TODO Auto-generated method stub
14                  return null;
15          }
16
17          @Override
18          public int compareTo(OverlayID o) {
19                  // TODO Auto-generated method stub
20                  return 0;
21          }
22  }
```

Dependent on our overlay specification, we have to define the type we need to store our concrete ID in the newly created class *MyOverlayID*. Lets say we want to create networks with a maximum node count of 2 Billion. So we can use the primitive integer to uniquely identify nodes with an ID. We can also use a primitive array representing our ID or the *BigInteger* class, which can nearly save every big number as long as your computers RAM is big enough ☺. Choosing the primitive integer results in a representation of a 32 Bit ID. In other words our network can scale up to $2^{32} - 1 = 4,294,967,295$. If we are going to use a primitive integer, then we need to ask ourself how to do all those calculations and comparisons between two integer, since they only represent a simple number in base 10. The best way is to store this ID number in a BigInteger. BigInteger represents an universal integer storage type. Universal means with no size limitation. But in our case, we only want to use BigInteger because of its fine methods presented by this class. So we are able to do complex binary comparisons and precise bit manipulations. This will save a lot of work.

On the Listing 3.2 you see two methods *getBytes* and *getUniqueValue*. The first is in most cases only interesting if your going to serialize your OverlayID object to send it over sockets. The second method is used to return the concrete unique value stored in our MyOverlayID class, this means the integer or BigInteger object you are using to store the ID.

Listing 3.2: An example how we could implement our ID object in MyOverlayID

```
1   public class MyOverlayID implements OverlayID {
2
3           //the id object holding the current id in space
4           private BigInteger myID;
5
6           public MyOverlayID(BigInteger myID) {
7                   //the new id must be generated and passed to our constructor
8                   this.myID = myID;
9           }
10
11          public MyOverlayID(int myID) {
12                  this.myID = BigInteger.valueOf(myID);
13          }
14
15          public byte[] getBytes() {
```

```
16                  //By default, we do not need this method
17                  //but anyway we use this for demonstration
18                  byte[] buf = null;
19                  try {
20          // Serialize to a byte array
21          ByteArrayOutputStream bos = new ByteArrayOutputStream() ;
22          ObjectOutput out = new ObjectOutputStream(bos) ;
23          out.writeObject(this);
24          out.close();
25
26          // Get the bytes of the serialized object
27          buf = bos.toByteArray();
28                  } catch (IOException e) {}
29
30                  //buf returns the exact size of this whole MyOverlayID
                        object
31                  return buf;
32          }
33
34          public Object getUniqueValue() {
35                  //return the id object
36                  return this.myID;
37          }
38
39          public BigInteger getValue() {
40                  return this.myID;
41          }
42
43          public int compareTo(OverlayID arg0) {
44                  MyOverlayID m = (MyOverlayID)arg0;
45                  return this.myID.compareTo(m.getValue());
46          }
47
48          public int getDistance(MyOverlayID id) {
49                  BigInteger oid = id.getValue();
50                  return this.myID.subtract(oid).abs().intValue();
51          }
52
53          public boolean equals(MyOverlayID o) {
54                  return this.myID.compareTo(o.getValue())==0;
55          }
56  }
```

On the Listing 3.2 we see some methods additionally defined. One method is *getValue()*, it has the same code like *getUniqueValue()*. The reason is to make it easier to get the concrete ID object. In this case, our ID object is a *BigInteger* object. Another solution for this would be to define the ID object as an *Integer*, which extends Object too and can be easily used for comparing and other complex operations. The second method is *getDistance(MyOverlayID id)*. Calculating the distance between two given ID's is in most cases a very important process for the routing core. Deciding which ID is closer to one

documents ID in an overlay network clarifies the problem.

After creating our OverlayID class, we need to create the contact class, which holds the newly created OverlayID and the IP address object. PeerfactSim.KOM holds an implementation of the IP address object, which we need to use as our IP address to send messages to other nodes. This address object is called *TransInfo*, representing an interface and has already its implementation called *DefaultTransInfo*. In the following Listing we have an example class called *MyOverlayContact*.

Listing 3.3: Example: The class MyOverlayContact

```
public class MyOverlayContact implements OverlayContact<MyOverlayID>,
    Comparable<MyOverlayContact> {

        private MyOverlayID myID;
        private TransInfo myTransInfo;
        private boolean isAlive;

        public MyOverlayContact(MyOverlayID id, TransInfo transInfo) {
                this.myID = id;
                this.myTransInfo = transInfo;
        }

        public MyOverlayID getOverlayID() {
                return this.myID;
        }

        public TransInfo getTransInfo() {
                return this.myTransInfo;
        }

        public void setAlive(boolean alive) {
                this.isAlive = alive;
        }

        public boolean isAlive() {
                return this.isAlive;
        }

        public MyOverlayContact clone() {
                MyOverlayContact newCon = new MyOverlayContact(this.myID,
                    this.myTransInfo);
                newCon.setAlive(this.isAlive);
                return newCon;
        }

        public int getDistance(MyOverlayContact con) {
                return this.myID.getDistance(con.getOverlayID());
        }

        public int compareTo(MyOverlayContact o) {
                return this.myID.compareTo(o.getOverlayID());
```

```
40          }
41
42          public boolean equals(MyOverlayContact o) {
43                  return this.myID.equals(o.getOverlayID());
44          }
45  }
```

By default we need some more methods like comparison, clone and distance methods. Cloning objects is important to ensure consistency, because we're going to create a network with thousands of nodes all with their OverlayID and OverlayContact objects in our computers RAM. Not cloning objects in the important parts of our new overlay will definitely result in bad behaviours. Typically we should add the *Cloneable* interface to our OverlayContact class. The OverlayID class does not need to be cloned, because every id is unique in our simulation, but this is dependent from your concrete implementation.

To clarify the need of a components clone ability, we see that *MyOverlayContact* from Listing 3.3 contains a flag called *isAlive*, which represents the current contacts alive state from the perspective of one node holding this contact. If a node wants to send this contact to another, it has to use the clone method to create a new contact object with a resetted alive flag.

The *getTransInfo()* method is used to get the IP address object of one active node in the current active network. Also the distance calculation (*getDistance(MyOverlayContact con)*) is needed for comparison cases.

3.5 The Routing Table

Next we are going to "implement" the routing table used by every node in our new overlay. In the first steps, before we implement our concrete routing table, we need to decide how to store our OverlayContacts to use it efficiently in the routing/sending mechanism. The simplest way to store our contacts is an array of MyOverlayContacts. But dependent on our overlays specification we possibly need to create a complex contact storage. It is also important to decide what kind of storage type we want to use. In some cases it could be interesting to use some collection class like LinkedList or Vector to store our OverlayContacts. In order to use as low memory as possible, in this tutorial we use a simple array of MyOverlayContacts. In the following Listing is a simple example with an implementation of the given interface OverlayRoutingTable, which must be parameterized to define the concrete elements to be stored in it.

Listing 3.4: Example: An automatically generated class MyOverlayRoutingTable in Eclipse with a simple storage example

```
1 public class MyOverlayRoutingTable implements OverlayRoutingTable<
      MyOverlayID, MyOverlayContact> {
2
3         private MyOverlayContact[] myRT;
4         private int size;
5
```

```
6         public MyOverlayRoutingTable(int size) {
7                 this.size = size;
8                 this.myRT = new MyOverlayContact[size];
9         }
10
11        public void addContact(MyOverlayContact contact) {
12                if(size() == this.size)
13                {
14                        //need to check for replacement
15                }
16                else
17                {
18                        //add it
19                        for(int i=0; i<this.size; i++)
20                        {
21                                if(this.myRT[i] == null)
22                                {
23                                        this.myRT[i] = contact;
24                                        return;
25                                }
26                        }
27                }
28        }
29
30        public int size()
31        {
32                int counter=0;
33                for(int i=0; i<this.size; i++) if(this.myRT[i] != null)
                      counter++;
34
35                return counter;
36        }
37
38        public List<MyOverlayContact> allContacts() {
39                List<MyOverlayContact> list = new ArrayList<MyOverlayContact
                      >(0);
40                for(MyOverlayContact con : this.myRT) list.add(con);
41
42                return list;
43        }
44
45        public void clearContacts() {
46                this.myRT = new MyOverlayContact[this.size];
47        }
48
49        public MyOverlayContact getContact(MyOverlayID oid) {
50                for(int i=0; i<this.size;i++)
51                {
52                        if(this.myRT[i].getOverlayID().equals(oid))
53                        {
54                                return this.myRT[i];
```

```
55                              }
56                      }
57
58                      return null;
59              }
60
61              public void removeContact(MyOverlayID oid) {
62                      for(int i=0; i<this.size;i++)
63                      {
64                              if(this.myRT[i].getOverlayID().equals(oid))
65                              {
66                                      this.myRT[i] = null;
67                                      return;
68                              }
69                      }
70              }
71      }
```

Typically we need to know the size of our routing table. In most cases, the size of the routing table depends on the size of our OverlayID and sight-range of our node. A simple scenario consists of a network with nodes having 50 entries in their routing table. The conditions for the correct entries in the routing table are protocol specific. Lets say using a 1-dimensional array of MyOverlayContacts is for saving entries by distance. This means the first entry in our array has the smallest distance to the holder of this routing table and the last entry the biggest distance.

In the Listing 3.4 we see exactly 6 methods, 5 of them must be implemented given by the interface. These methods are standards to manage the content of our new routing table. Additionally we need some more methods for comparison, cloning, distance calculation and many others, which we need in our routing mechanism and other processes.

3.6 Messages

All messages that our nodes are going to send in the simulations must be implemented. Typically its only important to define the message types. This means, messages are used like commands in an overlay sent to other nodes. Some messages contain some objects like the routing table or some other objects that needs to be exchanged between nodes. All messages that we are going to implement must extend *AbstractOverlayMessage*. This class has to be parameterized, to concretize its type of the OverlayID used by all messages. The best way to create the message classes is to create one overlay-specific abstract message class extending *AbstractOverlayMessage*, then creating the concrete message classes extending the newly created overlay-specific message class. This gives us the ability to define some general methods in our abstract class needed in all messages, like hop count and overlay-specific informations.

In Listing 3.5 we see our abstract OverlayMessage defined to use for our new messages:

Listing 3.5: Example: MyOverlayMessage as an abstract message class

```
public abstract class MyOverlayMessage extends AbstractOverlayMessage<
    MyOverlayID> {

        private int hopCount = 0;

        public MyOverlayMessage(MyOverlayID sender, MyOverlayID receiver) {
                //saving the sender and receiver in the super class
                super(sender, receiver);
        }

        public Message getPayload() {
                return this;
        }

        //this must be calculated by the subclasses
        public abstract long getSize();

        //returning current hop count in a simulation
        public int getHopCount() {
                return this.hopCount;
        }

        //used in the routing process
        public void incHop() {
                this.hopCount++;
        }
}
```

After creating our abstract message class, we start to create the concrete message classes. The following listing shows one example of a message class called *SimpleMessage*, representing a simple message, which can be sent between nodes:

Listing 3.6: Example: A simple message class MySimpleMessage

```
public class MySimpleMessage extends MyOverlayMessage {

        public MySimpleMessage(MyOverlayID sender, MyOverlayID receiver) {
                super(sender, receiver);
        }

        @Override
        public long getSize() {
                //sum the size of the components in this message
                //class and return
                int sendersize = getSender().getBytes().length;
                int recvsize = getReceiver().getBytes().length;
                return sendersize+recvsize;
        }
}
```

The most important message classes are the join and maintenance specific messages.

Above all is the join message, which contains initiation and preparation-specific information of the node and its routing table.

3.7 The Node Implementation

The most important component in our overlay, the node, has the base methods, in most cases the base routing core and many other implementations to run properly in an overlay. In our example we will create a class called *MyOverlayNode*. First we will check what kind of methods we have to implement. Then we discuss the needed methods and algorithms we should implement in our new node.

Listing 3.7: Example: The base node class MyOverlayNode

```
1 public class MyOverlayNode extends AbstractOverlayNode {
2
3         public MyOverlayNode(OverlayID peerId, short port) {
4                 super(peerId, port);
5         }
6
7         @Override
8         public TransLayer getTransLayer() { ... }
9
10        public void connectivityChanged(ConnectivityEvent ce) { ... }
11 }
```

On the first line we see MyOverlayNode extending AbstractOverlayNode. This abstract class provides all base components like the routing table and the translayer object, needed for message transmissions. By default, the base algorithms like the routing mechanism and routing table specific methods should be implemented in that node.

In line 8 of Listing 3.7 we have the method which returns our TransLayer, this object is needed for message transmissions. It provides the methods to send messages to other nodes using their addresses from the *OverlayContact* object.

This node must handle all message incomes and message replies. To avoid an unclear node structure with many message handlings, we should create a central class, which handles all messages incoming. All components awaiting messages or message replies must implement the interface *TransMessageCallback* (for message replies) and *TransMessageListener* (listening on one port for incoming message requests). So we create a new message handler and set it to listen on a port for new messages. The following listing shows an example of that message handler:

Listing 3.8: Example: The message handler for listening on incoming messages

```
1 public class MyMessageHandler implements TransMessageCallback,
    TransMessageListener {
2
3         private MyOverlayNode node;
4
5         public MyMessageHandler(MyOverlayNode node)
```

```
6        {
7                //set our node to the handlers pointer
8                this.node = node;
9        }
10
11       public void messageTimeoutOccured(int commId)
12       {
13               //handle message timeouts
14               //here we should try to resend that message
15               ...
16       }
17
18       public void receive(Message msg, TransInfo senderInfo, int commId)
19       {
20               //here we get a reply from a node with address senderInfo
21               ...
22       }
23
24       public void messageArrived(TransMsgEvent receivingEvent)
25       {
26               //here we get new messages arriving
27               //The TransMsgEvent contains all objects like message and
                     commId
28               ...
29       }
30 }
```

This message handler must be registered as an Listener on messages. This step is done by getting the TransLayer in the node and calling *addTransMsgListener* to register our new message handler. After doing this, our node is ready to receive new messages. The following listing clarify this:

Listing 3.9: Example: Registering the message handler for listening on new messages

```
1  public class MyOverlayNode extends AbstractOverlayNode {
2
3        private MyMessageHandler messageHandler;
4
5        public MyOverlayNode(OverlayID peerId, short port) {
6                super(peerId, port);
7
8                //create the message handler
9                this.messageHandler = new MyMessageHandler(this);
10
11               //add the message handler to the listeners list
12               this.getTransLayer().addTransMsgListener(this.messageHandler
                     , this.getPort());
13       }
14
15       ...
16 }
```

New messages arriving on our node results in calling the method *messageArrived*. In this method we must differentiate the message types and handle them. An example handling those messages is listed as follows:

Listing 3.10: Example: Handling new messages incoming on the node

```
1       ...
2
3       public void messageArrived(TransMsgEvent receivingEvent)
4       {
5               Message msg = receivingEvent.getPayload();
6
7               if (msg instanceof JoinRequest) {
8                       processJoin(receivingEvent);
9               } else if (msg instanceof Leave) {
10                      processLeave(receivingEvent);
11              } else if (msg instanceof Ping) {
12                      processPing(receivingEvent);
13              } else if (msg instanceof Anything) {
14                      ...
15              }       else { ... }
16      }
17
18      ...
```

3.8 The Operations

To properly run overlay-specific operations in our simulations, we have to create our operation classes extending *AbstractOperation*. One operation always represents one command like join or leave. All operations contain a method called *execute*, which is used to start the operation. Starting one operation should be always done by using the method *scheduleImmediately()*. Later on, we will concretize this step.

In the following Listing we see a simple operation, which shows its concrete structure implementing from *AbstractOperation*:

Listing 3.11: Example: A simple operation class SimpleOperation

```
1 public class SimpleOperation extends AbstractOperation<MyOverlayNode, Object
    > {
2
3       protected SimpleOperation(MyOverlayNode component, OperationCallback
            <Object> callback) {
4               super(component, callback);
5       }
6
7       @Override
8       protected void execute() {
9               //doing something like sending messages
10              //or maintainig the root node, etc.
```

```
11          }
12
13          @Override
14          public Object getResult() {
15                  //this operation has all methods we need to check
16                  //its state and get information, so we return THIS.
17                  return this;
18          }
19  }
```

3.8.1 Operation callbacks

In order to be informed about operation completion, we have to use *OperationCallback*. Every time an operation gets finished, it could be interesting to be informed if the operation was successful or not. Lets say we use a maximum execution time of 5 seconds for an operation. This means we schedule the operation with the method *scheduleOperationTimeout(long timeout)*. Reaching an execution time of 60 seconds for the operation results in calling the method *operationTimeoutOccured()*. This would sign the unsuccessful finish of the operation and can be catched by referencing to an *OperationCallback*.

The best way to "observe" the finish state of all operations running on a node, is to create a class, which acts as a central operation callback listener. In the following listing you will see a simple example how to do that.

Listing 3.12: Example: Using the OperationCallback listener on operations

```
1  public class MyOperationListener implements OperationCallback<Object> {
2          private MyOverlayNode _masterNode;
3
4          public OperationListener(MyOverlayNode masterNode) {
5                  this._masterNode = masterNode;
6          }
7
8          public void calledOperationFailed(Operation<Object> op) {
9                  //Operation op has failed
10                 //handle it
11         }
12
13         public void calledOperationSucceeded(Operation<Object> op) {
14                 //Operation op was successful
15                 //handle it
16         }
17 }
```

Listing 3.13: Example: Using the OperationListener for callbacks

```
1  //in MyOverlayNode we create an operation
2  //and assign the OperationListener to it
3
4  MyOperationListener myOpListener = new MyOperationListener(this);
5
```

```
6 MySimpleOperation simple_op = new MySimpleOperation(this, myOpListener);
7 //5000000 means 5 seconds on the simulator timer
8 simple_op.scheduleOperationTimeout(5000000);
```

3.9 The Bootstrap Part

To be able to let nodes create a network with new nodes joining to that network, we have to create a central list containing all active nodes. This list, called the *BootstrapManager*, represents the only central part in an overlay, which is a must to give nodes the ability to join the network. Since a new node does not know where to join the network, it must be able to access some kind of address book of active nodes, in order to become an active part of a peer-to-peer network. Choosing one of these nodes to send a join-request to, is the first step of the join-process of every node starting to join.

Listing 3.14: Example: The BootstrapManager

```
1 public class MyBootstrapManager implements BootstrapManager {
2
3         private List<OverlayNode> activeNodes;
4
5         public MyBootstrapManager()
6         {
7                 this.activeNodes = new LinkedList<OverlayNode>();
8         }
9
10        public List<TransInfo> getBootstrapInfo()
11        {
12                //put all TransInfo objects from activeNodes
13                //in a new list and return
14                List<TransInfo> list = new LinkedList<TransInfo>();
15                for(OverlayNode cNode : this.activeNodes)
16                        list.add(((MyOverlayNode)cNode).getLocalContact().
                              getTransInfo());
17
18                return list;
19        }
20
21        public void registerNode(OverlayNode node)
22        {
23                this.activeNode.add(node);
24        }
25
26        public void unregisterNode(OverlayNode node)
27        {
28                this.activeNode.remove(node);
29        }
30 }
```

3.10 Applications

Applications, running on top of the node, must be registered on the nodes, they are running on top of them. One application is set to one node. It is also possible to assign many applications to one node. So every node must hold a reference to an (array of) application(s) running on top of the node. PeerfactSim.KOM provides the needed interface, called *AbstractApplication*. The best way is to create an abstract application class, called *MyOverlayMessage*. In the Listing below we see one example:

Listing 3.15: Example: An abstract application class

```
public abstract class MyOverlayApplication extends AbstractApplication {

        private MyOverlayNode node;

        public MyOverlayApplication(MyOverlayNode node) {
                this.node = node;
        }

        protected MyOverlayNode getNode() {
                return this.node;
        }

        //called when an application-specific message arrives
        public abstract void deliver(MyOverlayMessage msg);

        //called when node joined or left the network, just to inform the
            application
        public abstract void update(MyOverlayContact contact, boolean joined
            );

        //any other methods
        public abstract void anyMethod();
}
```

On Listing 3.15 we see two methods *deliver* and *update*. These two methods are part of the Common API for structered peer-to-peer overlays presented in [1]. The first method is used if the node receives a message which is application specific. This message is then delivered to the application by upcalling this method. Also the method *update* informs the application about nodes joining or leaving the network.

Listing 3.16: Example: A simple application class

```
public class MyApplication extends MyOverlayApplication {

        public MyApplication(MyOverlayNode node) {
                super(node);
        }

        @Override
        public void anyMethod() {
```

```
9                 //...
10        }
11
12        @Override
13        public void deliver(MyOverlayMessage msg) {
14                //a message arrives for the application
15                //handle it
16                System.out.println("New message arrived: "+msg);
17        }
18
19        @Override
20        public void update(MyOverlayContact contact, boolean joined) {
21                //the current node state changed
22                //node joined or left the network
23                System.out.println("Node "+contact+" joined? "+joined);
24        }
25
26        //a simple method to send application specific
27        //messages to other applications
28        public void send(MyOverlayMessage msg) {
29                this.getNode().send(msg);
30        }
31 }
```

3.11 Component factories

The two most important component factories will be the application factory and the node factory. Both of them are used by the simulator to create these components for the simulations. Since nodes and application are from type Component, automatically defined by extending AbstractApplication and AbstractOverlayNode, we need to fully initialize these components in our factory. The following two listings show an example implementation of the two needed factories. On the next subchapter we will create the configuration file which gives the simulator all needed informations to run a simulation properly.

Listing 3.17: Example: Component factory for creating nodes

```
1 public class MyOverlayNodeFactory implements ComponentFactory {
2
3         //we use port 123 for msg transmissions
4         private short port = 123;
5
6         public MyOverlayNodeFactory() {
7         }
8
9         public Component createComponent(Host host) {
10                //create the new OverlayNode and set host to that new node
11                //return new node
12                MyOverlayNode newNode = new MyOverlayNode(newOverlayID(),
                      port);
```

```
13                 return newNode;
14         }
15
16         public MyOverlayID newOverlayID()
17         {
18                 //create new OverlayID with random class
19                 //getting seed from the simulator
20                 MyOverlayIDFactory fac = MyOverlayIDFactory.getInstance();
21                 return fac.createNewID();
22         }
23 }
```

Listing 3.18: Example: Component factory for creating applications

```
1 public class MyOverlayApplicationFactory implements ComponentFactory {
2
3         private short port = 123;
4
5         public MyOverlayApplicationFactory() {}
6
7         public Component createComponent(Host host) {
8                 //create application
9                 //here we can also create a new node and pass it to the
                      application
10                MyOverlayApplication newApp = new MyApplication(newNode(host
                      ));
11                newApp.setHost(host);
12                return newApp;
13        }
14
15        public MyOverlayNode newNode(Host host) {
16                MyOverlayNode newNode = new MyOverlayNode(newOverlayID(),
                      port);
17                return newNode;
18        }
19
20        public MyOverlayID newOverlayID()
21        {
22                //create new OverlayID with random class
23                //getting seed from the simulator
24                MyOverlayIDFactory fac = MyOverlayIDFactory.getInstance();
25                return fac.createNewID();
26        }
27 }
```

3.12 The Routing Mechanism

Every peer-to-peer protocol has its routing mechanism. Dependent on our protocol specification we have to decide how we implement the routing mechanism in our node to provide good routing qualities. Implementing the routing core brings important things together,

like the routing table structure, the quality of the routing table (good entries), the maintenance of our routing table and the repair-ability after node failures or nodes leaving the network without notification.

By default the routing core should implemented within the node class. It is much clearer to encapsulate the routing core in a class and reference it to the node.

In the following Listing 3.2 we see a simple message routing process. By default the receiver of a message first checks if the message is at its destination, which means that the message needs to be delivered to the current node or application. If not, it has to be routed to another node. This is done by getting the "best" next contact from the routing table with a specific search and compare algorithm.

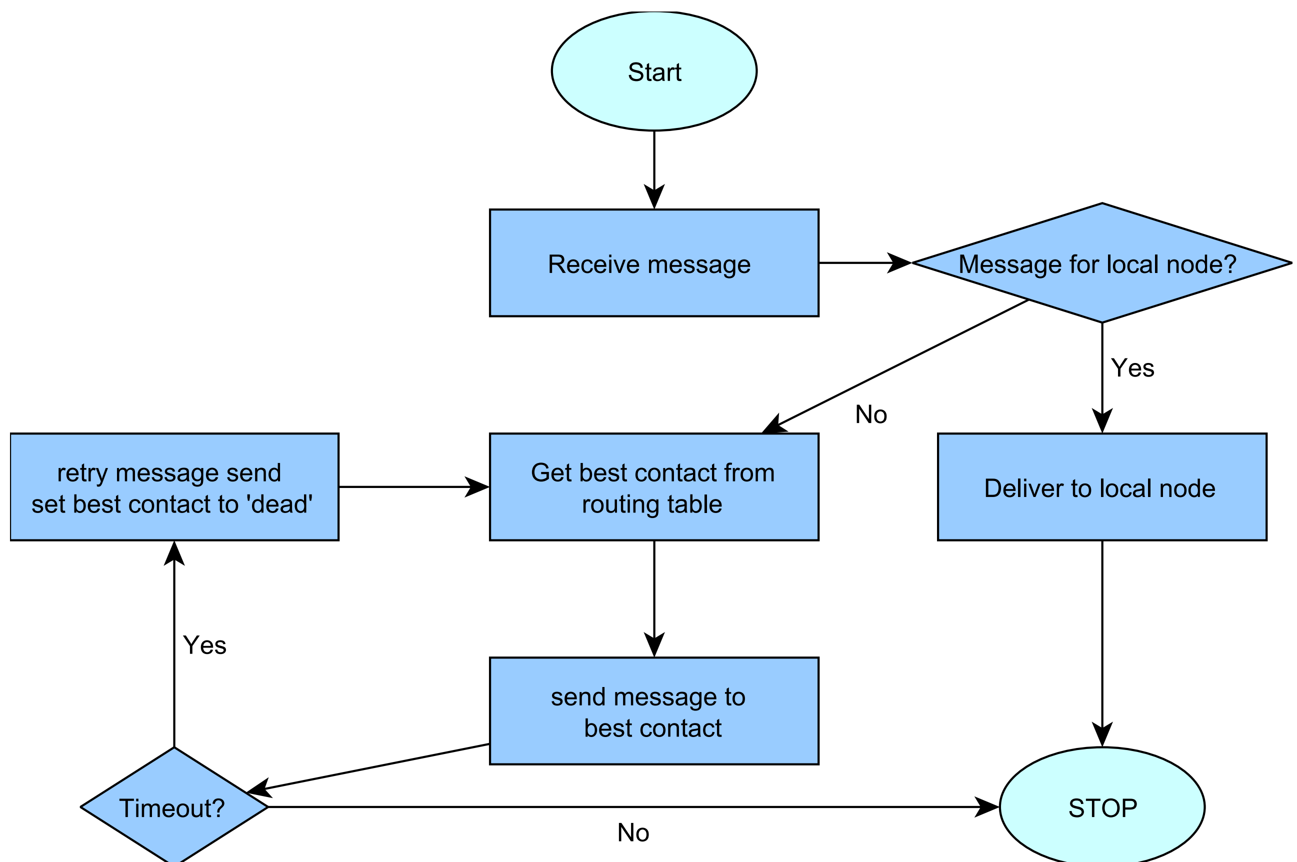

Figure 3.2: A simple message routing process

3.13 Creating networks with my overlay

There are at least two steps we have to do, before we can start simulations with our new overlay. The first is to create a configuration file which configures the simulator and gives it the ability to create all needed components like applications and nodes to run properly.

The next step is to define the action file. The action file contains commands, which are also parsed by the simulator and passed to the nodes or applications. Since the configuration file defines the components needed for a simulation and their parameters, the action file holds the commands passing them to the components during a simulation. By default we only need to use the application factory for our simulations, since we create the node and assign it to the application in the application factory.

First we will see an example configuration file. After that, we describe every entry in that XML configuration file step-by-step:

Listing 3.19: Example: A simple configuration file

```
1 <?xml version='1.0' encoding='utf-8'?>
2   <Configuration>
3     <Default>
4       <Variable name="seed" value="786876" />
5       <Variable name="finishTime" value="6000m" />
6     </Default>
7
8     <SimulatorCore class="de.tud.kom.p2psim.impl.simengine.Simulator" static
          ="getInstance" seed="\$seed" finishAt="\$finishTime" />
9
10    <NetLayer class="de.tud.kom.p2psim.impl.network.simple.SimpleNetFactory"
          downBandwidth="200" upBandwidth="100">
11      <LatencyModel class="de.tud.kom.p2psim.impl.network.simple.
            SimpleStaticLatencyModel" latency="10"/>
12    </NetLayer>
13
14    <TransLayer class="de.tud.kom.p2psim.impl.transport.
          DefaultTransLayerFactory"/>
15
16    <ComponentFactory class="de.tud.kom.p2psim.example.MyOverlayNodeFactory"
          />
17
18    <Monitor class="de.tud.kom.p2psim.impl.common.DefaultMonitor" start="0m"
          stop="\$finishTime">
19      <Analyzer class="de.tud.kom.p2psim.example.MessageAnalyzer"/>
20    </Monitor>
21
22    <HostBuilder class="de.tud.kom.p2psim.impl.scenario.DefaultHostBuilder"
          experimentSize="1000">
23      <Host groupID="peer1">
24        <NetLayer/>
25        <TransLayer/>
26        <ComponentFactory />
27      </Host>
28
29      <Group groupID="group1" size="200">
30        <NetLayer/>
31        <TransLayer/>
32        <ComponentFactory />
33      </Group>
```

```
34
35        <Group groupID="group2" size="799">
36          <NetLayer/>
37          <TransLayer/>
38          <ComponentFactory />
39        </Group>
40      </HostBuilder>
41
42      <Scenario class="de.tud.kom.p2psim.impl.scenario.CSVScenarioFactory"
            actionsFile="path/to/actions.dat" componentClass="de.tud.kom.p2psim.
            example.MyOverlayNode">
43      </Scenario>
44  </Configuration>
```

Now we describe the configuration file from Listing 3.19. In Listing 3.19, we see a simple configuration file. This file is the main configuration file to run a simulation properly. In this file the simulator gets all information it needs to initiate its simulation.

In the first part, we see the *Default* Tag. This is needed for defining variables. In line 4 and 5 we see two well defined variables. The first holds the current seed used by the simulator and our factory using this seed for java's randomizer. This seed is useful since we can reproduce the same ID space in our simulations. The second variable holds the time to finish the whole simulation. 6000m means 6000 minutes. These time values do not represent the real time running on your system. Its an internal virtual time used and simulated by the simulator itself.

The next part is the definition of the Simulator core we want to use for our simulation. By default we use the class in *de.tud.kom.p2psim.impl.simengine* and pass it the two defined variables. This tag must be named *SimulatorCore*.

On the next part we are going to define the underlying components in our simulation we need. The first is the network layer we want to work on. Since PeerfactSim.KOM abstracts the first four layer of the OSI-model [5], we need to define the layer provided by the simulator we want to use for our simulation. We call this part the *NetLayer*. In this tag we define the factory class of type *ComponentFactory*, which is able to create the network layer of type *AbstractNetLayer*. In this case we use the *SimpleNetFactory*, which creates a simple network layer with simple routines. We also pass two variables to define the total bandwith for a node. For every network layer we have to define a latency model, which will be used to calculate the latency while transmitting messages. In this case we use a simple static latency model. Please check the package for other network layers and latency models.

The following part is most important, because its referencing to our new node factory. The Tag called *ComponentFactory* defines the class from which the simulator will get all the nodes needed to create the simulation. The Simulator will automatically detect it as a *ComponentFactory* and will call the method *createComponent* to create the nodes.

Then we can add the *Monitor* Tag to define Analyzers we want to use for analyzing our overlay during a simulation. These analyzers are important for evaluation purposes. In Chapter *The Evaluation part* we will discuss how we create those analyzers and how

they will work. First, we pass the *start* and *stop* time for the analyzers. By default we begin monitoring from 0 to the *finishTime* value. Then we pass the analyzers one by one. In our example we have one analyzer called the *MessageAnalyzer*.

Now we define the hosts and groups the Simulator is going to create after creating our nodes with the given node factory. This part is statically called *HostBuilder*. We created one node and two groups of nodes. The reason we create groups of nodes is dependent on our action file. The action file is equal to a script file holding the commands for the nodes they must execute in a simulation. To create a single node we need to define a *Host* tag, for groups we need to define *Group* tags. In addition to these tags we need to pass unique names for these Hosts and Groups. The attribute called *groupID* stands for the unique name. For defining groups of nodes we need to pass a size value for the group. This attribute causes the simulator to group nodes to one group, passing a command to that group results in passing this command to all nodes in this group.

In the last part we define the scenario type. There are two scenario types we can use in the current version of PeerfactSim.KOM (v3.0). The first is *CSVScenarioFactory* and the second is *DOMScenarioFactory*. The first needs a parameter called *actionFile*. This file contains all commands line by line, which will be executed on the nodes referenced to in that action file. The second scenario type is able to read the commands directly within the *Scenario* Tag. The scenario tag needs an attribute called *componentClass*, which references to the component acting as a host in the simulation. In this case we reference to *MyOverlayNode*. This causes the simulator to pass all commands in the actions part to the component of type *MyOverlayNode*, which is already well defined as *ComponentFactory* in the configuration file. In the following two Listings you will see those two scenario types, which you can use for simulation.

Listing 3.20: Example: The actions file

```
1 #join network
2 peer1 1m join
3 group1 2m-50m join
4 group2 40m-150m join
5
6 #lets stay idle
7 peer1 500m idle
8 group1 510m idle
9 group2 530m idle
10
11 #now lets send some messages
12 peer1 630m sendmessage
13 group1 640m sendmessage
14 group2 660m sendmessage
15
16 #all nodes leaving now the network
17 peer1 800m leave
18 group1 820m leave
19 group2 840m leave
```

Listing 3.21: Example: The actions directly read in from configuration file by DOMScenarioFactory

```
<Scenario class="de.tud.kom.p2psim.impl.scenario.DOMScenarioFactory">
  <Action groupID="peer1" time="1m">join</Action>
  <Action groupID="group1" time="1m">join</Action>
  <Action groupID="group2" time="1m">join</Action>
  <Action groupID="peer1" time="2m">idle</Action>
  <Action groupID="group1" time="2m-3m">idle</Action>
  <Action groupID="peer1" time="3m">sendmessage</Action>
  <Action groupID="group1" time="3m-5m">sendmessage</Action>
  <Action groupID="peer1" time="4m">leave</Action>
  <Action groupID="group1" time="7m-9m">leave</Action>
  <Action groupID="group2" time="10m">leave</Action>
</Scenario>
```

Typical commands in the actions file are join, idle, sendmessage and leave. But how does the node get this command and execute it? In order to do that, we implement methods in the component class with names equal to the commands in the acion part. This means that we need to create a method in our component MyOverlayNode, called *join*, calling results in initiating the join operation. Also the commands idle, sendmessage and leave must be implemented.

3.14 Evaluation of the new Overlay

In this section we will discuss how to create an analyzer and how they will work in a simulation. In Listing 3.19 we saw only one analyzer tag referencing to *MessageAnalyzer*. This analyzer will be automatically detected by the scenario parser before the simulation is started. The fact is, all analyzers will be hooked automatically into the simulator core before it starts. This means that the simulator will notify all analyzers on the important parts of a simulation like message transmission, operation initiation and similar points. Depending on what kind of analyzer we define, our analyzers will be hooked automatically.

There are four different analyzer types we can use. These are *TransAnalyzer*, *NetAnalyzer*, *OperationAnalyzer* and *ChurnAnalyzer*. In the following we describe those analyzers in detail (Please check comments in java files):

- **TransAnalyzer:** TransAnalyzers receive notifications when a network message is sent, or received at the transport layer. This means if we send a message from our node, then a method will be called, which must be implemented in our analyzer implementing *TransAnalyzer*. Also receiving messages on our node calls a method in *TransAnalyzer*. Every time a node sends a message, the *transMsgSent(AbstractTransMessage msg)* method is called on the *TransAnalyzer*. Even if a message arrives on a node, the *transMsgReceived(AbstractTransMessage msg)* method will be called on the *TransAnalyzer*.

- **NetAnalyzer:** NetAnalyzers receive notifications when a network message is send, received or dropped at the network layer. This type of analyzer is similar to *Trans-*

Analyzer but referencing to a lower layer of the simulator. When a message gets transmitted from one host to another on the network layer, one of the two methods *netMsgSend(NetMessage msg, NetID id)* and *netMsgReceive(NetMessage msg, NetID id)* will be fired, notifying about a message transmission on the network layer. There is a third method *netMsgDrop(NetMessage msg, NetID id)*, which will be called if a message is going to be dropped in case of a non-existent receiver for this message.

- **OperationAnalyzer:** OperationAnalyzers receive notifications when a operation is finished either with or without success. With this analyzer we can hook every operation initation and completion. One example is to catch the join operation after finishing, in order to be informed when to check the node for its state. This analyzer is the most used one in evaluations. When an operation gets executed directly or by the simulator, the method *operationInitiated(Operation<?> op)* will be fired, signalizing an operation initiation. Also finishing an operation is by default called manually. This means we could lastly call the method *operationFinished(boolean success)* in the *execute* method or after receiving some reply in our operation. This depends on the type of our operation.
- **ChurnAnalyzer:** ChurnAnalyzers receive notifications when the network connectivity has been changed of churn affected hosts. Everytime one hosts connection status changes, the method *onlineEvent(Host host)* or *offlineEvent(Host host)* will be called. These two methods signalize the connection state on the network layer. Like your internet connection could lag or crash for a while, in a simulation the method *offlineEvent(Host host)* would be called to signalize that your host have no connection to the network.

Before we discuss the *MessageAnalyzer* class we look at its content:

Listing 3.22: Example: Showing one analyzer for analyzing messages

```
1 public class MessageAnalyzer implements Analyzer, TransAnalyzer {
2
3         private String receiveInfo="";
4         private String sendInfo="";
5
6         public MessageAnalyzer() {}
7
8         public void start() {
9                 //this method is called when when analyzer
10                //is started by the simulator
11        }
12
13        public void stop(Writer output) {
14                //is called after finishing the simulation or the analyzer
                      itself.
15                //here we should do some output, like writing in files,
16                //writing to console.
17                //object 'output' is referenced to the console.
```

```
18                  try
19                  {
20                          output.write("Finished message analyze\n");
21                          output.write("Results:\n");
22                          output.write(this.receiveInfo);
23                          output.write(this.sendInfo);
24                  }
25                  catch(IOException ex) { /* error */ }
26          }
27
28          public void transMsgReceived(AbstractTransMessage msg) {
29                  //we add the new arrived message to our string object
30                  this.receiveInfo+= "New message received: "+msg.getPayload()
                        +"\n";
31          }
32
33          public void transMsgSent(AbstractTransMessage msg) {
34                  //we add the sent message to our string object
35                  this.sendInfo+= "New message sent: "+msg.getPayload()+"\n";
36          }
37  }
```

Every analyzer we create, must implement the *Analyzer* interface. It provides the two methods *start()* and *stop()*. The first signalizes the start of this monitor, the second method is called after finishing the simulation or reaching the monitors *stop* time.

In *MessageAnalyzer* we want to catch up all messages received and sent by all nodes. While the simulation is running, we put all message informations into Strings and print them to the console when the *stop* method is called. Another way would be the creation of files to create diagram specific files (e.g. GNUPlot) to do better evaluation.

The following Listing 3.23 shows where to add the correct tags for our new monitor *MessageAnalyzer*.

Listing 3.23: Example: Adding the analyzer correctly to the configuration file

```
1 <Monitor class="de.tud.kom.p2psim.impl.common.DefaultMonitor" start="0m"
     stop="\$finishTime">
2    <Analyzer class="de.tud.kom.p2psim.example.MessageAnalyzer"/>
3 </Monitor>
```

We suggest to create the analyzer, implement the analyzer interface and all needed analyzer types. Then reference to this analyzer by adding the *Analyzer* Tag to the *Monitor* Tag in your configuration file.

Chapter 4

FAQ

This chapter answers some common and implementation based questions.

4.1 Common questions

Q: Is it possible to create full functional applications running on the simulator?
Yes, you can create any kind of peer-to-peer based application running on the simulator.

4.2 Implementation

Q: Can i run overlays without operations?
Yes, it is possible to run commands without encapsulating them in operation classes. Since the simulator calls the methods defined in the action file, its enough to implement those methods executing all commands during a simulation. Operations in PeerfactSim.KOM are created according to the Command-Design Pattern [2].

Q: Do i have to use all interfaces presented in this tutorial to create simple network based applications in PeerfactSim.KOM?
No, the only things you have to use are the components for the network layer. When we are going to send messages in our applications, we have to use the transport layer, which we have to "create" and then use. Every component sending messages has to be created as a host using the network layer and transport layer to send messages to other hosts. So every application sending messages have at least a fully initialized host component. All messages sent by these hosts must be from type *Message*.

Q: Im using OperationCallbacks but none of the two methods *calledOperationFailed* and *calledOperationSucceeded* gets called even if my operation gets started or finishes. Why?
These two methods get only called if you call the method *operationFinished(boolean success)*. This method signalizes that your operation is finished successfully or not. For example in an operation you send a message and wait for a reply. After you get the

reply you call *operationFinished(true)* which calls *calledOperationSucceeded* on the OperationCallback. When you get a *messageTimeoutOccured* while waiting for a reply you should call the method *operationFinished(false)* results in calling *calledOperationFailed*.

Q: I want to create an analyzer for my overlay which counts all messages sent by all hosts. How do i do that?
The first step is to create the analyzer class. To sign the class as an analyzer, you have to implement it as defined in the interface *Analyzer*. The second is to decide which type of analyzer we want to get a counter on the messages sent by the hosts. This let us use the *TransAnalyzer* Interface for the analyzer class which informs us about message transmissions on the transport layer. So our class needs to implement this interface. Third, we have to summarize all messages sent by the host on our TransAnalyzer. When the simulation or the monitor gets finished the method *stop* gets called, which gives us the ability to put the result out by passing the result to the *Writer* object which is referenced to the console. Fourth, we need to modify the configuration file and add the Monitor tag with the analyzer tag referencing to our new analyzer class. The following Listings show these steps in detail.

Listing 4.1: Example: A simple message counter analyzer

```
1  public class MessageAnalyzer implements Analyzer, TransAnalyzer {
2
3          private int messages = 0;
4
5          public void start() {
6                  //this method gets started when analyzer
7                  //is started by the simulator
8          }
9
10         public void stop(Writer output) {
11                 //here we should do some output, like writing in files
12                 //writing to console
13                 //object 'output' is referenced to the console
14                 try
15                 {
16                         output.write("Finished message analyze\n");
17                         output.write("Messages sent: "+messages+"\n");
18                 }
19                 catch(IOException ex) { /* error */ }
20         }
21
22         public void transMsgReceived(AbstractTransMessage msg) {
23                 //we ignore message arrives
24         }
25
26         public void transMsgSent(AbstractTransMessage msg) {
27                 this.messages++;
28         }
29 }
```

Listing 4.2: Example: Adding the Analyzer to the configuration file

```
1 <Monitor class="de.tud.kom.p2psim.impl.common.DefaultMonitor" start="0m"
    stop="\$finishTime">
2         <Analyzer class="de.tud.kom.p2psim.overlay.pastry.analyzers.
              MessageAnalyzer"/>
3 </Monitor>
```

Q: How do i pass parameters to the commands used in the actions file? All commands used in the actions file, are able to receive primitive parameters. This means you can pass integers and strings easily. Since all commands used in the actions file must be implemented as methods on the main component running in the simulation, these methods must accept those parameters. Passing objects to the commands must be done by using the *Parser* Interface. Lets say you want to pass an object holding a string object to a command, the first is to create the parser class to parse the string value and create the object. The Simulator detects automatically the parameter type your command methods accepts and uses the correct parser to create the parameter object to pass it to the command. The following Listing shows this in detail.

Listing 4.3: Example: Parsing objects to commands

```
1  public class MySimpleParser implements Parser {
2
3          //getType() helps the simulator to detect
4          //the parameter type to pass to the command
5          public Class<ParameterObject> getType() {
6                  return ParameterObject.class;
7          }
8
9          public ParameterObject parse(String value) {
10                 return new ParameterObject(value);
11         }
12 }
13
14 public class ParameterObject {
15         private String val;
16
17         public ParameterObject(String val) {
18                 this.val = val;
19         }
20
21         public String getValue() {
22                 return this.val;
23         }
24 }
```

Now modify the configuration file by adding the ParamParser tag to it.

Listing 4.4: Example: Adding the Analyzer to the configuration file

```
1 //passing the ParamParser tag to the Scenario tag in the
2 //configuration file results in registering it as a parameter
3 //parser for the commands used in the actions file.
```

```
4 <Scenario class="de.tud.kom.p2psim.impl.scenario.CSVScenarioFactory"
    actionsFile="\$actions">
5         <ParamParser class="de.tud.kom.p2psim.example.MySimpleParser"/>
6 </Scenario>
```

Implement the command in your component which accepts this new parameter type.

Listing 4.5: Example: Passing objects to commands

```
1 public void hello(ParameterObject object) {
2         System.out.println(object.getValue());
3 }
```

The command which will be parsed by the *MySimpleParser* class and then passed to the *hello* method.

Listing 4.6: Example: The command passing parameter objects

```
1 #the command hello passes the string value HelloWorld to the method
2 #hello with parameter ParameterObject which is parsed by the
3 #MySimpleParser class.
4
5 peer1 1m join
6 peer1 2m hello HelloWorld
7 peer1 3m leave
```

Bibliography

[1] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. *Proc. of IPTPS*, 3, 2003.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[3] http://peerfact.kom.e-technik.tu darmstadt.de/. Peerfactsim.kom. *The discrete event based P2P Simulator written in Java*, 2006.

[4] M. Ripeanu and I. Foster. Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems. *First International Workshop on Peer-to-Peer Systems (IPTPS)*, 68, 2002.

[5] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *Communications, IEEE Transactions on [legacy, pre-1988]*, 28(4):425–432, 1980.

List of Figures

3.1 A simple ring topology represented by integer ID's 12
3.2 A simple message routing process 28

Listings

3.1 Example of OverlayID . . . 12
3.2 Putting id object and methods in MyOverlayID . . . 13
3.3 Example implementation of OverlayContact . . . 15
3.4 Example implementation of OverlayRoutingTable . . . 16
3.5 Example implementation of MyOverlayMessage . . . 18
3.6 A simple message class for an overlay . . . 19
3.7 The base overlay node . . . 20
3.8 The message handler for listening on messages . . . 20
3.9 Registering the message handler for listening . . . 21
3.10 Handling message incomes . . . 22
3.11 A simple operation class . . . 22
3.12 A simple operation callback listener . . . 23
3.13 Use OperationListener . . . 23
3.14 The BootstrapManager . . . 24
3.15 The abstract application class for our applications . . . 25
3.16 The simple application class . . . 25
3.17 The component factory class for creating nodes . . . 26
3.18 The component factory class for creating applications . . . 27
3.19 A simple configuration file . . . 29
3.20 The CSVScenarioFactory example . . . 31
3.21 The DOMScenarioFactory example . . . 32
3.22 The MessageAnalyzer example . . . 33
3.23 Configure the Analyzer . . . 34
4.1 Analyzer: Message Counter . . . 36
4.2 Adding the analyzer to the configuration file . . . 36
4.3 Analyzer: Parsing objects to commands . . . 37
4.4 Adding the analyzer to the configuration file . . . 37
4.5 Command getting objects as parameters . . . 38
4.6 The command passing parameter objects . . . 38